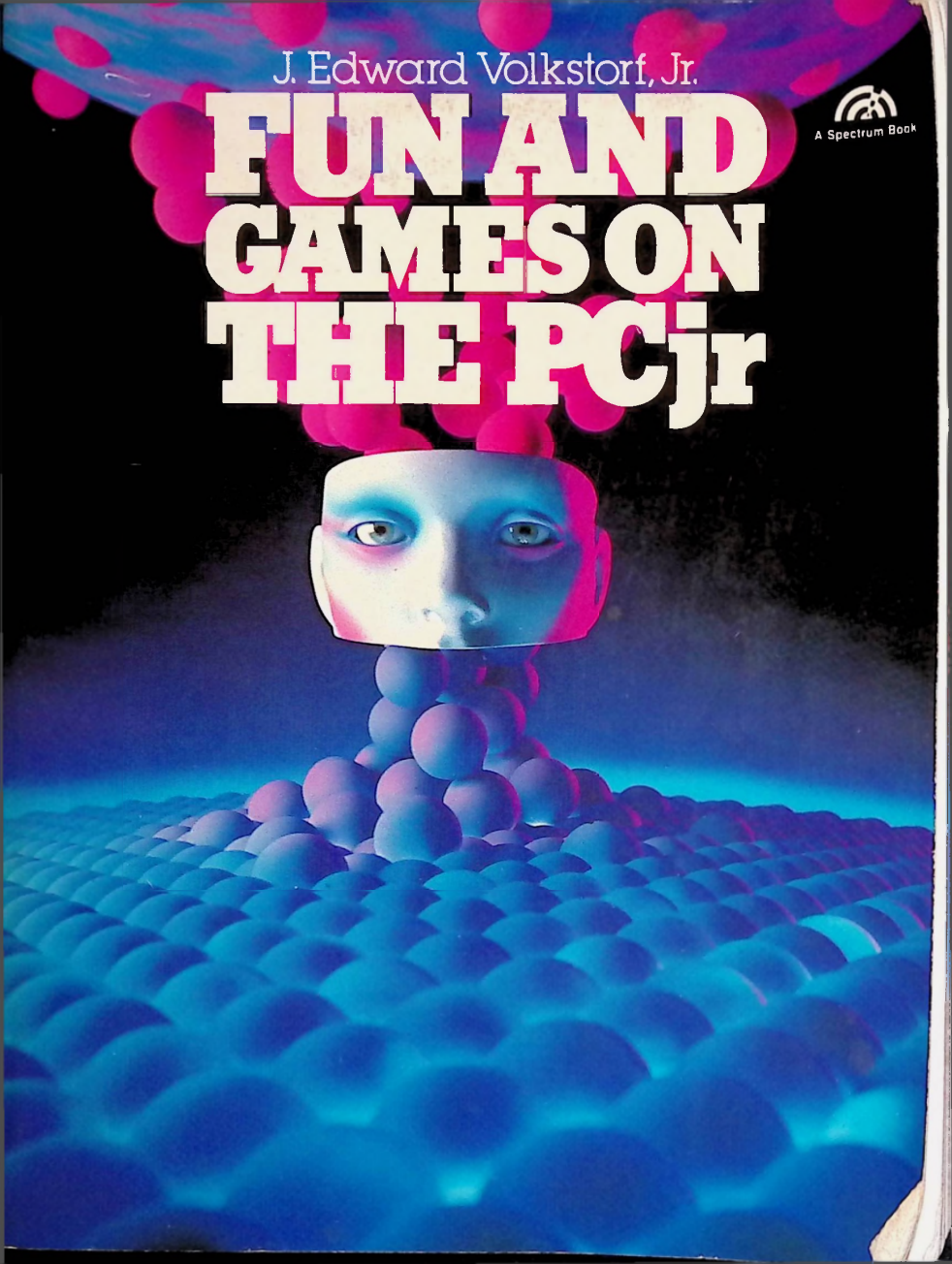
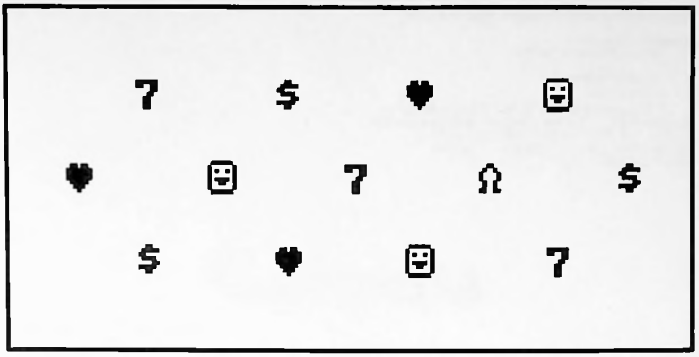
J. Edward Volkstorf, Jr.

FUN AND GAMES ON THE PCjr



A Spectrum Book

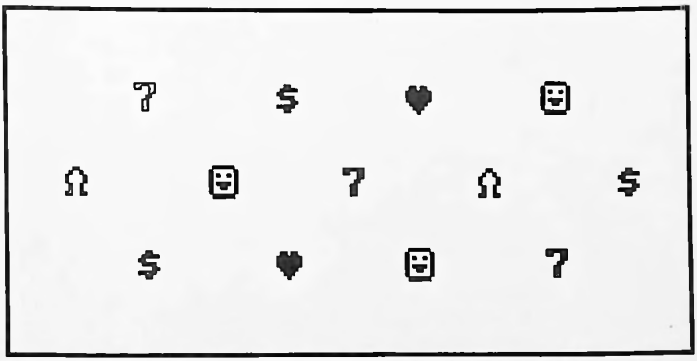


J. Edward Volkstorf, Jr.

FUN AND GAMES ON THE PCjr

A Spectrum Book

Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632



Library of Congress Cataloging in Publication Data

Volkstorf, J. Edward.
Fun and games on the PCjr.

"A Spectrum Book."
Includes index.
1. Computer games. 2. IBM PCjr (Computer)—Programming. 3. Basic (Computer program language)
I. Title. II. Title: Fun and games on the P.C. jr.
GV1469.2.V64 1985 794.8'2 85-3383
ISBN 0-13-332487-7 (pbk.)

©1985 by Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632.

All rights reserved. No part of this book may be reproduced in any form or by any means without permission in writing from the publisher.

A SPECTRUM BOOK

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-332487-7

Editorial/production supervision by Marilyn E. Beckford
Chapter openings by Alice R. Mauro
Cover design by Hal Siegel
Cover photo by Michel Tcherevkoff
Manufacturing buyer: Carol Bystrom

Bookware® is a registered trademark of Prentice-Hall, Inc.

This book is available at a special discount when ordered in bulk quantities. Contact Prentice-Hall, Inc., General Publishing Division, Special Sales, Englewood Cliffs, N.J. 07632.

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books Limited, *Wellington, New Zealand*
Editora Prentice-Hall do Brasil Ltda., *Rio de Janeiro*

Contents

Introduction 1

1
Anatomy of a Game 5

2
"I Guess It's . . ." 25

3
The Word Is . . . 47

4
An Adventure Game 73

5
Deal 'em 91

6
Casino Playtime 131

7
Board Games 155

8
Action Games 183

9
Educational Games 225

10
Wrap-Up 245

Index 277

Preface

Here we go! It's another book on computer games using the BASIC language, but this book has a more modern, colorful, and thorough coverage than any books before it. You have probably seen the many books filled either with BASIC code (from cover to cover) or with a combination of supportive text and the code that provides some examples of gaming programs. Invariably, these books contain most (if not all) printer-based or TTY-like (for teletype printer) games that target their programs to everybody with a micro and BASIC. Thus, the inherent capabilities for graphics that many micros have are ignored in favor of a more widespread readership.

This leads to the first two goals of this book. First, all the games herein are graphics-oriented and use standard Microsoft BASIC as found on the IBM PC, PCjr, and the many PC "clones." Second, the subject of computer gaming is discussed in a logical and concise framework that organizes games into a number of categories, each supported by one or more example programs. Along the way, a number of software techniques useful in gaming programs (as well as in many other types of software) are also developed.

But this is not all . . . . Writing good gaming software can be a complex task. Coding the types of data being processed, the subtleties in the speed and timing of the program, and the need to be user friendly and still tolerant of user indiscretions (for example, not crashing with unusual inputs), and other problems put quite a burden on the gaming programmer. Understanding and learning to write program logic with these thoughts in mind will add to your base of programming skills and your bag of coding tricks.

A final goal is simply having fun and adding to your software library. Each example program is a complete, well-developed game, not the trivial "print your name 50 times on the screen"–type of program that some books call "games." Most of the programs in this book are short enough for you to

key them in without any major difficulty. Or there is a PC/PCjr–compatible diskette containing the programs that can be purchased from me that alleviates the need for your typing each of the programs. (See appendix B for more details on this.)

ACKNOWLEDGMENTS

Just a few . . .

Hats off to all the gaming fanatics, computer-based or otherwise, to whom this book is dedicated, and to Sandy, Chris, and Todd for putting up with yet another book.

Introduction

People have played games in one form or another since the dawn of time. Indeed, some archeological digs have uncovered what appear to be crude precursors of six-sided dice. Imagine some caveman losing his favorite axe to a roll of the dice (or "baby needs a new pair of saber-tooth fur-lined shoes"?). Since then, there have come into being playing cards, board games, pencil and paper games, and on and on. Gambling is one of the more popular forms of game playing as well as a profession for some persons. Probability and statistics began as studies of the outcomes of various gambling games.

There are also gaming theory, simulations, models, and management games. And there is the advent of the computer. A plethora of computer-based video and arcade games has virtually overnight become a multibillion-dollar industry.

Mankind has and probably always will find the time to partake of a little friendly competition. Maybe this whole state of affairs is based on our inherent need to compete. Could it be that in the future whole nations may play games of a sort? (Are they doing it now?)

If you glanced at the table of contents, you probably noticed that each chapter, except the first and last ones, are categories of types of games. We first introduce some key computer gaming concepts in Chapter 1 by means of a single program example that will be dissected in some detail. A number of different classifications are then developed for computer games as a whole to form the basis for the remaining chapters.

In each chapter, the major elements of the games developed are considered first. The role of the computer, some programming techniques, and other miscellaneous programming thoughts are discussed. Following this discussion are the one or more sample game

programs to put the ideas into action. Further descriptions follow, illustrating the specifics of each program.

The sample programs have been kept fairly short (from about 100 lines to the largest of about 300). Each sample program follows a number of standards such as:

1. All programs include a code that allows them to be called via a menu program or started directly via a RUN command.

2. Each program normally exits to a section of code that begins at line 9000, which handles returning the user to the menu program or back to BASIC. Function key 1 is an "abort" key, which may be pressed at any time to cause an early exit from the program (that is, without playing it to an orderly end point).

3. The name of each program as provided by the author on the optional diskette is given in the first line, number 0, of each program. These four- to six-character names are also used to identify the programs when they are referenced in the text.

4. Single-character inputs are used wherever possible to simplify user interaction with the games. This is done by using the INKEY$ function. In particular, many programs use the convention of asking the user to press the space key (or space bar) to proceed through the games whenever a pause is needed.

5. Any multicharacter input required of the user, such as betting amounts in the card games, uses the LINE INPUT statement preceded by a message of the form:

```
Enter ante amount:
```

where the word "Enter" is always used to specifically signify that the Enter key must be pressed after the information indicated is keyed.

6. For numeric input, a LINE INPUT N$ expression is always used followed by converting the value in the N$ variable to the appropriate numeric value via the VAL function. This circumvents BASIC'S printing error messages of its own when erroneous input is made by the user.

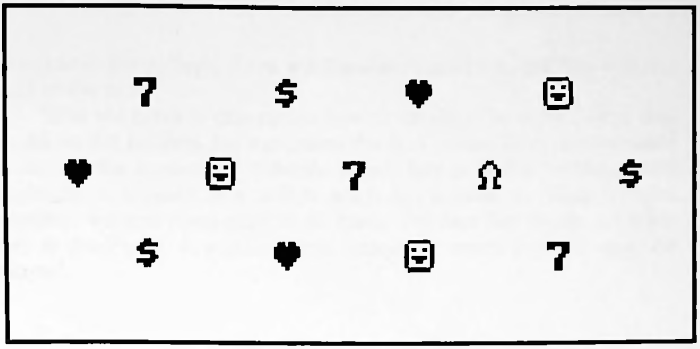
7. All programs are written in sections indentified by a one-line remark preceded and followed by a blank line so that the remark stands out from the rest of the program.

8. Last, there are remarks after most lines of code in the programs to further aid the user in understanding the program logic used. BASIC allows the apostrophe to be used versus REM as a lead in or indicator of a comment—thus: 100 INPUT x: REM gets number
100 INPUT x' gets number.

For the most part, instructions for playing each example game should be obvious on the basis of the prompts made by the program. There is also a short section of remarks at the beginning of each program that briefly explains the nature of each game. In all cases, complete details on playing the games are contained in the text of the book.

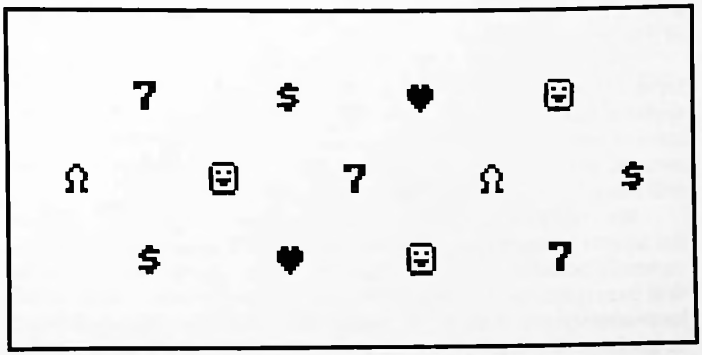
All chapters end with a section on suggested variations for each of the sample programs provided. There are hints and thoughts on how to expand the programs with additional playing options and other embellishments. Readers will most likely develop many variations on their own.

A set of appendices closes the book, providing material for reference. Included is a section on random numbers which provides two sample programs for graphing random-number generators. There is also an appendix on writing a menu program, tying together all the games. Now, with a sense of direction, step right this way. Chapter 1 and your first game, PC Says, beckons....



1

Anatomy of a Game



Let's design a game for our IBM PC. To begin, we need an idea upon which to base the game. Instead of trying to come up with something completely original, we will borrow an idea from another game. Everybody is probably familiar with "Simon Says." For instance,

"Simon says...read the title of this chapter."
"Simon says look at the first sentence on the next page."
"Read the page number you are now on."

If you read the title for this chapter and the first sentence on the next page, then good. But if you looked down at the page you were on, then, uh oh...Simon didn't say so! The general idea is to repeat some action but only when Simon says to do it. Also, if the actions are sufficiently complex, then maybe you'll foul up trying to do them, even if Simon said so. Now let's consider a computer-based version of this game.

What we will do is let the PC generate a pattern that we will have to repeat. Since the PC can't talk (at least not in the variety of systems that most of us own and use), we have to let the pattern involve some actions that require using the keyboard. Thus, our pattern will simply be a particular sequence of keystrokes that the player will have to repeat based on what the PC has displayed.

We could just pick a series of alphabetic letters, print them on the screen momentarily, then let the player try to rekey them. This is certainly possible, but play like this is more akin to a touch-typing drill than a game. So let's limit the keystrokes required to some smaller related group such as the ten function keys, the numeric keypad, or perhaps the four cursor control keys. With only four keystrokes

possible in the pattern, there will be some repetition, but this will just add to the fun.

Now we have to determine how to display the cursor keys that make up the pattern. Let's recreate the four cursor keys as oversized boxes on the screen. To indicate which key is in the pattern, we'll highlight a shaded box within each key's interior. With a color monitor, we can even color-code them. For one last touch, as each key is displayed, a musical note unique to each key will also be played.

THE GAME PC SAYS

Our game is as follows: Four boxes are displayed on the screen in the same pattern as the four cursor control keys. The PC will then display them in some random sequence by highlighting their interiors and sounding a musical note associated with each one. After the PC's pattern has been displayed, the player must rekey the same cursor-key sequence. If keyed properly, the player scores a few points. We will call the game PC Says.

It's time to begin designing the actual program code. One of the most important parts of a computer game is the screen layout—a well-developed screen is like window dressing. A simple or even poor game idea can be made fairly interesting if the graphics used in the game display enough originality. As this is our first game, and a relatively simple one at that, we will keep its graphics fairly simple.

The four boxes for the four cursor-control key symbols can each be displayed as a group of block characters using the character code 219. Let each group be composed of four rows with nine characters per row. Since there are four such groups to print a subroutine that prints them at a specified row and column, it is not necessary for us to code each group separately. The subroutine is as follows:

```
820 '
830 ' print 9 x 4 pad at R,C
840 '
850 FOR J=R TO R+3
860 LOCATE J,C : PRINT STRING$(9,219);
870 NEXT : RETURN
```

The variables R and C will indicate where on the screen the four-by nine-character group or box is to be displayed. Note how the

J variable in the FOR loop counts how many rows of characters to print and also serves as the actual row position in the LOCATE statement.

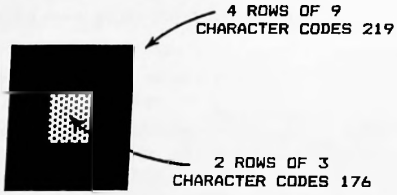
To highlight a particular box, we can just print an insert within the box in the form of a smaller group of block characters but in a different color from the other boxes. If you have a monochrome display and adapter, this means you will not be able to see the highlighted areas. To get the best of both worlds, we can use a different character code with a color statement. That way, monochrome users can still distinguish which key has been activated, and color-adapter users can enjoy the game in color.

The character codes 176 to 178 are shading characters and will serve our purposes nicely. We will use number 176, which has the least shading, in a two- by three-character group (two rows and three columns). The same R and C variables used in the previous subroutine can also be used here, but they must be offset by 1 and 3, respectively, so the insert is centered properly. Since there are four keys to highlight, a subroutine such as the above is also appropriate. Note that we forego the FOR NEXT loop since only two rows of characters are printed.

```
880 '
890 ' print 2 x 3 insert at R+1,C+3
900 '
910 COLOR K: LOCATE R+1,C+3: PRINT STRING$(3,176);
920 LOCATE R+2,C+3 : PRINT STRING$(3,176);
930 COLOR 7: RETURN
```

See Figure 1.1 for an example of what the box with a highlighted interior looks like.

FIGURE 1.1. *Highlighted pad in a box from the game PC Says.*



Our next step is to calculate the pattern that the PC plays. This involves using a random generator, which is supplied by BASIC's RND function. The pattern consists of a sequence of the four keys that we can code as simply the numbers 1 to 4. To generate the random values the statement:

```
KS(I)=INT(RND*4)+1  ' create and display the random sequence
```

The RND function returns a value between 0 and just less than 1. Multiplying this value by 4 gives a value of 0 to just less than 4, which becomes 3 because of the INT function (the INT function drops digits after the decimal point). By adding 1 to this value, the range of numbers becomes 1,2,3,and 4. Note that we have no idea whether the pattern will be repeated or which numbers will appear. It is just as likely that a pattern length of 3 can be the numbers 1, 2, 3 as well as 2, 2, 2 and anything else using the four numbers. The RND function is used in most of the games featured in this book. Appendix A covers the RND function in more detail and also offers some background material on other types of random numbers.

At this point, we can display the block characters for the cursor control keys, highlight them, and have a random value to display. Now it's time to tie these steps together. First map each of the sequence numbers into a unique keystroke and assign the musical note that will sound for each group. Each cursor control key must also have a specific position on the screen for the block characters that represent it.

We will leave the top third of the screen for our score and other such messages. This gives us rows 8 to 24 for our four keys. The cursor-up and cursor-down keys will be centered in this area, with the left and right cursor keys in the middle and to the left and right of the remaining screen area. Picking the actual row and column positions is often a matter of just trying one place over another until the overall appearance is balanced and reasonbly pleasing to the eye: This is one more reason for coding display subroutines with variables for the row and columns. It makes it easier to change things if they don't look right by simply using different values for the variables.

For our four boxes, the following positions on the screen will be used:

Cursor Key	Row	Column	Musical Note
Left	15	26	D
Up	10	36	C
Right	15	46	E
Down	20	36	F

The note column is the actual musical note that is played when the key is pressed. It can be easily changed in the program to any single note you desire or even to multiple notes played in sequence.

The row and column values will be used initially to position each box that represents the four cursor control keys. As the game is played and a random sequence is generated, these same table values will be used to position the inserts within each box to indicate that a particular cursor key has been chosen. The note for each key will also be played as the insert is displayed.

To map each of the numbers in the random sequence into the appropriate keystroke positions and musical notes, we *could* use a table (subscripted array); instead, we will opt for putting them in a series of subroutines as follows:

```
750 '
760 ' set values for each pad
770 '
780 R=15:C=26:N$="D":RETURN
790 R=10:C=36:N$="C":RETURN
800 R=15:C=46:N$="E":RETURN
810 R=20:C=36:N$="F":RETURN
```

An ON / GOSUB is used to set the values based on the keystroke numbers 1 to 4. Thus for key number 3, we effectively perform a GOSUB 800 to set that key's R, C, and N$ variables. The overall program code to signify a particular key from the random sequence array KS(I) is as follows:

```
520 FOR I=1 TO RS
530 KS(I)=INT(RND*4)+1  ' create and display the random sequence
540 ON KS(I) GOSUB 780,790,800,810: K=KS(I): GOSUB 910 : PLAY N$ : GOSUB 850
550 NEXT I
```

After Program 1.5 is executed, it is then the player's turn to try to repeat the pattern by pressing the appropriate cursor control keys.

Processing for this part of the game will consist of reading each keystroke made by the player, verifying that each is a valid key, and comparing it to the KS array to determine if it is the correct one for this

part of the random sequence. If it *is* correct, the next keystroke is retrieved and the process is repeated for the next element in the KS array. If the entire sequence is keyed correctly, the player scores some points. However, if any keystroke is incorrect, the program stops reading the keystrokes and informs the player that he has goofed.

The procedure for programming the steps described is as follows:

1. A WHILE / WEND loop will control reading each keystroke until all the keystrokes for this pattern are read or an error is made.

2. An INKEY$ statement will do the actual reading of each keystroke. The program code consists of a GOTO, which loops on C$=INKEY$ until a key has been pressed: This occurs when C$ is not equal to a null string.

3. Once a keystroke is made, the program then executes a SOUND 99,0 statement. This turns off the note that was played when the previous keystroke was made. This keeps musical notes that are played in sync with the pace at which the player presses each cursor key; otherwise, the program would wait for each note to be played in its entirety before playing the next note.

4. Since the cursor control keys are on the numeric key pad, a check will be made to determine if either a cursor control key or a number key has been pressed. A cursor control key is indicated by a C$ character string with a character length of two where the second character is a K, H, M, or P, corresponding to the number keys 4, 8, 6, and 2. Both of these character strings correspond to the left, up, right, and down cursor control keys. The INSTR function converts the cursor control keys to the numbers 1, 2, 3, and 4:

```
K=INSTR ("KHMP", RIGHT$(C$,1))
```

for the cursor keys, or

```
K=INSTR ("4862", C$)
```

for the numeric keystrokes. The variable K is used to select the boxes for highlighting and the note to be played for each key.

5. If a keystroke is made correctly, then K will equal the value in the random sequence array, KS, for this particular position in the

pattern. The program will then increase its counter for the keystrokes and continue the loop.

6. For an invalid keystroke, a flag (the variable OK) is set to be checked by the WHILE / WEND loop. If set, the loop stops.

7. When the main loop is concluded, the flag will be used to either increase the players score, that is, if the player entered each keystroke correctly, or the player can be admonished for making a mistake in a keystroke.

The complete program code for each of the above steps is:

```
580   WHILE I<=RS AND OK
590   C$=INKEY$ : IF C$="" THEN 590  ' wait for player's keystroke
600   IF LEN(C$)=2 THEN K=INSTR("KHMP",RIGHT$(C$,1))
                    ELSE K=INSTR("4862",C$) ' cursor or numeric key pressed?
610   IF K=0 THEN PLAY "04 C64":GOTO 650  ' ignore everything else
620   SOUND 99,0 : IF I>1 THEN GOSUB 850 ' clear previous note, and pad
630   ON K GOSUB 780,790,800,810 : GOSUB 910 : PLAY N$ ' player's note
640   IF K<>KS(I) THEN OK=0 ELSE I=I+1 ' is it correct?
650   WEND
```

Our game is just about complete—all that remains is keeping track of the score. Tallying the score is easy enough—just add some value to a counter—but determining what amount to add is a different matter. In general, you want to have the score reflect the level of difficulty of the game or the phase of the game being played at the time the points are scored. For PC Says, the scoring will be based on some measure of the difficulty of the random sequence that has to be repeated. Our approach will be to let the length of the sequence be the number of points that is scored if the sequence is entered properly.

This brings up another point about games in general. It's a good idea to create games that gradually increase in difficulty as they are played. In this way, they continue to present a challenge even as the player becomes more adept at playing them. For PC Says, our challenge will be to repeat ever-increasing lengths of random sequences. The game will begin with a random sequence of 3 characters and move on from there to lengths of 4, 5, 6, and so on. The final pattern length can be an arbitrary number, but over 8 or 9 becomes very difficult to repeat.

While we are developing the scoring mechanism, we might also consider adding a second player. This works particularly well with

PC Says. So instead of creating a solitaire game, we will keep track of two scores, one for each player. As one additional consideration in scoring, allow two or three turns per random sequence length instead of just one. This makes the difficulty level increase more gradually, allows higher scores, and offers a greater chance for a player to make a comeback in two-player games.

Incorporating these last few features is a matter of adding three FOR / NEXT loops to the code already developed. The first FOR / NEXT loop will count the number of the round currently being played and determine the length of the random sequence. The next loop will set the number of turns a round is to be played. The third and final loop indicates which player is playing. The programming for each of these loops is:

```
420 FOR RP=1 TO NR            ' loop for each round of play,
430 LOCATE 3,60:PRINT RP;     '   display round number
440 FOR T=1 TO TR             ' loop for TR turns this round
450 FOR P=1 TO NP             '   and once for each player

                - - -

700 NEXT P : NEXT T : NEXT RP ' next player, turn, and round
```

The variables NR and TR are set in the program at 4 and 2. A third variable, SL, will also be set for the starting length of the random pattern. The variable SL has an initial value of 3 and is used in calculating the length of the random pattern, which is variable RS. As each new round begins, RS is set based on this initial length plus the round being played. This causes the difficulty of the game to increase as the round being played increases. The variable NP is set to the number of players, 1 or 2, by the user when he starts the program.

The top third of our screen has been reserved for displaying the scores. Instead of a plain and simple printing of numbers, a character mode box is used to highlight the scores. (See Figure 1.2) Along with the scores, an indicator of which player is active, the round being played, and a logo that names the game will also be displayed. As each player takes his turn, a number (either 1 or 2) is highlighted by printing the word PLAYER and his number (P from the FOR loop) in inverse video, black on white. Note that when the player is finished, the message has to be reprinted to reset the screen back to normal video.

FIGURE 1.2. PC Says game-scoring area.

This same player—number variable will also index the two-element array S, which contains the player's scores. Thus, to display the score for player P, we simply code

```
PRINT S(P);
```

and to update it, we can code

```
S(P)=S(P)+RP
```

where RP is the round number being played.

At the end of our game, the program checks NP to determine if there was only one player; if so, the program will end. But if two players were competing, a message indicating who won and the final score can be printed. (This is just a matter of assuming that player 1 won, and setting P to this value. Then the scores are checked, and if player 2 was the actual winner, then P is reset to 2 and the scores in the S array are SWAPped (flip-flopped).) The possibility of a tie also has to be checked for. The BASIC program code for this is:

```
710 LOCATE 7,28:PRINT"Game over. "; : IF NP=1 THEN 740
720 P=1:IF S(1)<S(2) THEN P=2:SWAP S(1),S(2)
730 IF S(1)<>S(2) THEN PRINT "Player ";P;" wins ";S(1);" to ";S(2)
                  ELSE PRINT "It's a draw!!!"
```

In addition to the program code described so far, some messages have been added to control the overall flow of the game, such as when the program is ready to play a round and when it's a particular player's turn. The BASIC code creates a pause in line 670 after the score is updated (or after the "you goofed" tune is played). It is a straight-forward FOR / NEXT loop but also includes the statement C$=INKEY$ so that any extra keystrokes made by the current player can be cleared out or absorbed before the next player begins. This is

needed in case an extra key or two is pressed after the program has determined that an incorrect sequence was entered.

Our last consideration is how to end the game. Instead of simply coding an END statement, we will use a technique that allows for an "early exit" as well as a normal exit. Let function key 1 signify that we want to abort the game, regardless of what point we are in playing. This can be handled by using an ON KEY / GOSUB statement, which will trap to our exit subroutine. The subroutine will ask you if you do indeed want to exit from the game or would like to restart it. Pressing the Esc key will indicate an exit from the game; pressing the space bar will indicate the restart of the game.

You may want to combine all of your games into an overall system controlled by a menu program. If a menu is used then each game must determine whether it was initiated via the menu or directly by the user. This is important so that when the game is finished it knows whether or not to return to the menu program. We will use the variable MENU$ to indicate the manner in which a program is started. If MENU$ has the value "NO", then the game was called directly. On the other hand, if MENU$ is null (is equal to), then the MENU program started up the game that was just played. The overall code for this is as follows:

```
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

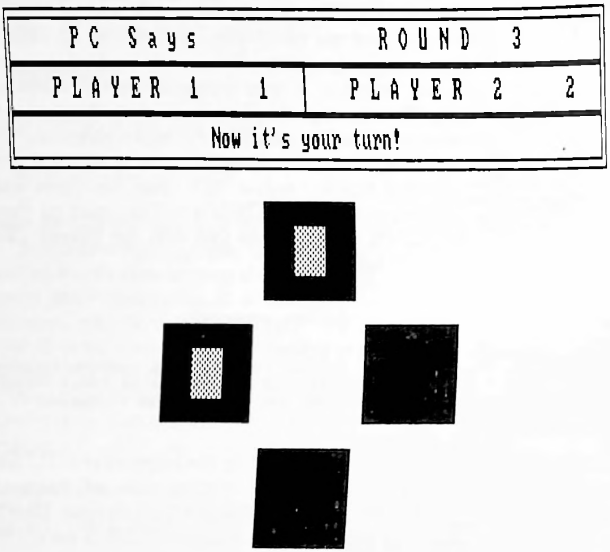
The variable MENU$ is set to "null" at the beginning of PC Says and at the start of all of the gaming programs. We will begin our games from the menu program via a CHAIN to line number 20 of the game selected, which is after the point where MENU$ is set to "NO" (at line number 10). Thus, our exit subroutine can tell how the current program was started since MENU$ will now have a null value.

When MENU$ is "NO", we exit back to the menu program; otherwise, we will simply END the program, leaving the user in BASIC. (See the menu program description in Appendix C for more details on how the menu program works.) It is not necessary to refer to Appendix C in order to use any of the gaming programs, but as you code more and more programs, the information in the appendix can be helpful in controlling your access to the games.

All of the game programs in this book make use of this entry and exit convention. To simplify the typing of this code (lines 9000 to 9070), you can enter it once into a file, save the file in an ASCII format, and then MERGE it into each of the programs. Lines 10 and 20 can also be included since they, too, are always the same in each game program.

Program 1.1, PC Says, is the complete program for our first game. (See Figure 1.3 for an example of it being played.)

FIGURE 1.3. PC Says game example.




```
 0 '  PCSAYS  PC Says - as in Simon Says
 10 MENU$="NO"
 20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
 30 '
 40 '  One or two players play and take turns trying to repeat the random
 50 '  pattern displayed on the screen by pressing the cursor control keys.
 60 '
 70 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
 80 DEFINT A-Z : RANDOMIZE VAL(RIGHT$(TIME$,2))
 90 PRINT"PC SAYS - Repeat the pattern by pressing the cursor control keys"
```

```
100 PRINT : PRINT "How many players (1 or 2)?";
110 NP=0 : WHILE NOT (NP=1 OR NP=2) : NP=VAL(INKEY$) : WEND
120 CLS
130 '
140 '  display borders for player numbers, scores, and other messages
150 '
160 LOCATE 2,10:PRINT CHR$(201);STRING$(58,205);CHR$(187) ' top
170 LOCATE 8,10:PRINT CHR$(200);STRING$(58,205);CHR$(188) ' bottom
180 FOR R=3 TO 7
190 LOCATE R,10:PRINT CHR$(186);:LOCATE ,69:PRINT CHR$(186) ' sides
200 NEXT
210 LOCATE 4,10:PRINT CHR$(199);STRING$(58,196);CHR$(182) ' the middle
220 LOCATE 6,10:PRINT CHR$(199);STRING$(58,196);CHR$(182) '  two lines
230 LOCATE 4,40:PRINT CHR$(194);  ' column to
240 LOCATE 5,40:PRINT CHR$(179);  '  separate the
250 LOCATE 6,40:PRINT CHR$(193);  '   player scores
260 LOCATE 3,17:PRINT"P C   S a y s";:LOCATE ,48:PRINT"R O U N D";
270 LOCATE 5,14:PRINT"P L A Y E R   1       0";
280 LOCATE 5,44:PRINT"P L A Y E R   2       0";
290 '
300 '  display the four pads
310 '
320 GOSUB 780 : GOSUB 850 : GOSUB 790 : GOSUB 850
330 GOSUB 800 : GOSUB 850 : GOSUB 810 : GOSUB 850
340 '
350 '  Initialize playing variables
360 '
370 NR=4 : TR=2 : SL=3  ' number of rounds, turns per round, sequence length
380 S(1)=0 : S(2)=0     ' player scores array
390 '
400 '  Now, play the game...
410 '
420 FOR RP=1 TO NR          ' loop for each round of play,
430 LOCATE 3,60:PRINT RP;   '   display round number
440 FOR T=1 TO TR           ' loop for TR turns this round
450 FOR P=1 TO NP           '   and once for each player
460 COLOR 0,7:LOCATE 5,(P-1)*30+14:PRINT" P L A Y E R  ";P;
470 LOCATE 7,24:PRINT" Press <SPACE> when ready to play ";
480 WHILE INKEY$<>" " : WEND ' wait for space key
490 LOCATE 7,24:COLOR 7,0:PRINT SPC(35);
500 RS=SL+RP-1 ' random sequence length increases by round number
510 PLAY "MF"  ' computer plays in foreground
520    FOR I=1 TO RS
530    KS(I)=INT(RND*4)+1  ' create and display the random sequence
540    ON KS(I) GOSUB 780,790,800,810: K=KS(I): GOSUB 910 : PLAY N$ : GOSUB 850
550    NEXT I
560 LOCATE 7,31:PRINT"Now it's your turn!" : OK=1 : I=1
570 PLAY "MB" ' player plays in background
580    WHILE I<=RS AND OK
590    C$=INKEY$ : IF C$="" THEN 590  ' wait for player's keystroke
600    IF LEN(C$)=2 THEN K=INSTR("KHMP",RIGHT$(C$,1))
                    ELSE K=INSTR("4862",C$) ' cursor or numeric key pressed?
610    IF K=0 THEN PLAY "O4 C64":GOTO 650  ' ignore everything else
620    SOUND 99,0 : IF I>1 THEN GOSUB 850 ' clear previous note, and pad
630    ON K GOSUB 780,790,800,810 : GOSUB 910 : PLAY N$ ' player's note
640    IF K<>KS(I) THEN OK=0 ELSE I=I+1 ' is it correct?
650    WEND
660 LOCATE 7,31 : IF OK THEN GOSUB 1020 ELSE GOSUB 970  ' check results
```

```
670 FOR I=1 TO 999 : C$=INKEY$ : NEXT : GOSUB 850     ' pause and clear chars
680 GOSUB 850 : COLOR 7,0:LOCATE 7,31:PRINT SPC(20); ' clear pad and message
690 LOCATE 5,(P-1)*30+14:PRINT" P L A Y E R  ";P;     ' reset background color
700 NEXT P : NEXT T : NEXT RP ' next player, turn, and round
710 LOCATE 7,28:PRINT"Game over. "; : IF NP=1 THEN 740
720 P=1:IF S(1)<S(2) THEN P=2:SWAP S(1),S(2)
730 IF S(1)<>S(2) THEN PRINT "Player ";P;" wins ";S(1);" to ";S(2)
                  ELSE PRINT "It's a draw!!!"
740 GOTO 9030 ' by-pass subroutines
750 '
760 ' set values for each pad
770 '
780 R=15:C=26:N$="D":RETURN
790 R=10:C=36:N$="C":RETURN
800 R=15:C=46:N$="E":RETURN
810 R=20:C=36:N$="F":RETURN
820 '
830 ' print 9 x 4 pad at R,C
840 '
850 FOR J=R TO R+3
860 LOCATE J,C : PRINT STRING$(9,219);
870 NEXT : RETURN
880 '
890 ' print 2 x 3 insert at R+1,C+3
900 '
910 COLOR K: LOCATE R+1,C+3: PRINT STRING$(3,176);
920 LOCATE R+2,C+3 : PRINT STRING$(3,176);
930 COLOR 7: RETURN
940 '
950 ' player's entry was wrong
960 '
970 PRINT"Whoops !!!   Sorry." : PLAY "P4 O2 D8. D8. C8.. O4"
980 RETURN
990 '
1000 ' player did it right
1010 '
1020 PRINT"Good going! Got it!" : PLAY "MS F16 E16 G8. G8.. G8 MN"
1030 S(P)=S(P)+RP  ' increment this player's score
1040 LOCATE 5,(P-1)*30+33 : PRINT USING"####";S(P);  ' and display it
1050 RETURN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart":C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 1.1 PCSAYS - PC Says game

Some Common Attributes of Gaming Programs

PC SAYS was chosen as our first game program because it is relatively short and involves most, if not all the programming considerations that the rest of our games will utilize. The next few paragraphs summarize these.

1. At its lowest level, PC Says does nothing more than generate a set of random numbers that the player must repeat. Yet by dressing up the program with boxes, scoring areas, and musical tunes, the game becomes playful fun—requiring a good memory with some skillful eye—hand coordination.

   The success of a game depends a great deal on how it is packaged and presented. With the IBM Personal Computer, there are many different graphics programming techniques that can be used in developing such presentations. A large part of this book will cover how such graphic representations can be made.

2. PC Says involves play that requires a certain degree of skill, ultimately reflected in the scores. On the other hand, there is little strategy and only a minor degree of chance involved. Some games developed in later chapters will require the player to utilize some strategy to play well while others will be based purely on random chance. This may sound like an indictment of such games, but it is not. A multibillion-dollar industry depends on such games: Except for blackjack, all casino games are (theoretically speaking) independent random events. Of course, their popularity is also tied to the potential for monetary reward and the free-wheeling atmosphere in which they are played.

3. The use of sound effects is similar to the graphics that dress up the visual appearance of the game. Clever tunes that are played during the course of a game can create more interesting play than a quiet, soundless game.

4. Progressive levels in a game also add much to the playing fun. For skill and strategy games, this is usually easy to incorporate in some form; but for those games based on pure chance events, the progression might be simply an increase in the points scored during the latter stages of play. Although no more "skill" than during earlier rounds is required, the interest and fun is heightened because of the prospect of more points.

Another approach to creating levels of play is to change the timing of the games—this is particularly prevalent in the animated games. As you increase in level of play, the speed and quickness of your adversary increases as well.

5. The role of the computer in games can take many forms. For PC Says, the computer serves as a "croupier" or referee. It keeps score and checks that we play by the rules of the game. In other games, it might be an adversary.

The computer can also be your partner in some games. Consider an educational game where the player must consider a number of options during the course of play. The computer could offer advice, help the player make the decisions, and even provide hints on playing.

6. In PC Says, the speed at which the program runs and the player responds was not considered. This feature could have been introduced in terms of how fast the player responds to the random pattern. (This is probably not necessary since most players will try to respond quickly, anyway, before they forget the pattern..)

In other games, the speed at which the game is played and the player reacts become the main attributes of the game. Using interpretive BASIC puts a limit on how effectively games such as these can be coded. In these games, compiled BASIC and other languages such as Pascal, FORTH, and, ultimately, Assembly Language can offer vast improvements. If the game is fairly simple, then reasonable speed can still be achieved with BASIC. This problem primarily affects action games involving the animation of many different images.

7. Another attribute to consider in any game is the learning experience (if any) it provides to the player. Some games are purely leisure-time diversions. But often a lesson in science, mathematics, and so on can be put in the form of a game. Even the improvement in eye-hand coordination that some games provide might be considered an educational experience. The experience of becoming familiar with the computer through a game provides some degree of computer literacy training.

For the most part, the games in this book will be of the pure fun variety. Chapter 9 covers educational games in more detail and develops two such games.

8. Scoring and player control were fairly simple in PC Says. The only information that needed to be maintained for each player was

his score. The score was stored in the two-element array S which was indexed by player number 1 or 2. Other games may have far more information regarding the status of play for each player.

For example, in card games, each player may have two or more cards, their bet, and current bankroll. In the animation games, there will also be a number of variables associated with each player. In general, multiplayer games will require keeping track of this data in one or more arrays, as in the PC Says scores. As long as the indexing is kept straight-forward, this should not present any problem to the gaming programmer.

If simple variables are used, such as S1 and S2 for the scores of player 1 and 2, respectively, then the programming can potentially be accomplished in the same manner; but as the number of players or other items maintained increases, this technique can quickly become very cumbersome.

In general, it is best to consider whether your game will be played by one or more than one player early on instead of when you have completed a single-player version. Retrofitting a one-player game to multiple players can be quite tedious.

Classifying Computer Games

Computer games can be divided into several categories or classifications. Some of these classifications have already been alluded to, such as strategy and skill games versus those that are based purely on random chance. Another breakdown might be action games and nonaction games; that is, games with or without animation. Other such classifications based on the overall type of game can also be used.

For the sake of organization, this book develops games based on the following categories:

1. *Guessing Games.* These games are based on a random data item that is chosen by the computer, and which the player must determine (guess). Each of these games involve a limited amount of skill and strategy.

2. *Word Games.* Word games require programs to manipulate *character text* in a particular manner in contrast to using *numerical values*, which is the style primarily used in our other programs.

Some of these games demonstrate how character strings are processed. Finding or guessing a word and word puzzles will also be developed.

3. *Card Games.* Playing card games on a computer requires some fairly sophisticated graphics that display the card images. The processing can also be tricky because of the coding of card values and what each card represents, as in a poker hand. A number of poker games will be programmed.

4. *Educational Games.* These games have an underlying goal of teaching the player something while he is having fun. Two such games will be developed.

5. *Animation.* Image movement about the screen introduces a number of new programming techniques not used in any of the previous games. A number of such games are programmed.

6. *Gambling Games.* Writing a program to simulate a real-life event often involves gamelike programming techniques. These programming techniques are applied to three casino games where the play is simulated by the program under the control of the player.

7. *Board Games.* Several popular board games are programmed in this chapter.

8. *Adventure Games.* An adventure game is like a word maze or interactive novel. It's also like entering another world or becoming another person. One such game is developed in this book that allows you to explore a haunted house.

PC Says Enhancements

Each chapter in this book closes with a section that offers suggestions for future development of its games—and PC Says is no exception. The following suggestions are offered as potential changes, improvements, or variations that you are invited to consider (you will probably come up with many more).

1. Change the game to three or four players. Expand the score array and set aside another area at the top of the screen for each additional player's scores.

2. Change the overall goal of the game so that the player must score a certain number of points in order to win. For example, in the two-player version, the first player to score 11 or more points wins.

3. Let the computer pick a random-length pattern each time, instead of progressively longer patterns. The pattern length could be limited to from four to seven items. For two-player games, be sure that both players have the same-length pattern for each turn. This change might also work well if included with the previous change.

4. In contrast with the above, you could have each player in a multiplayer game select the length of the pattern. If you get yours right, the next player must try to match a pattern of the same length; if you are wrong, then the next player must pick the next pattern length. This introduces a type of "challenge match" quality to the game.

5. When the program picks the random pattern, make sure no alike numbers are in succession in the random pattern.

6. Change the speed at which the pattern is displayed. This can be done by coding a variable note length command in a PLAY statement before the note is played. Decrease the length of the notes in the later stages of the game.

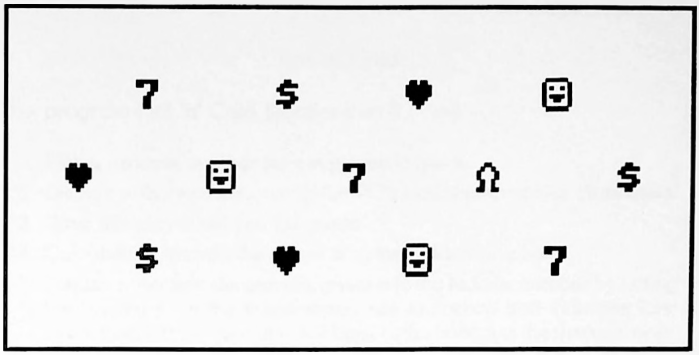
7. Give each player a second chance to guess the pattern if he has made a mistake. Then lower the score amount for the second chance. You could also have the game repeat the pattern before the second try.

8. In conjunction with the above, slow down the second pattern when played to give the player a better chance for a correct answer.

9. Penalize a player for an incorrect guess. Allow him to pass if he feels that a pattern is too difficult. This change in play can also be combined with the other scoring changes for even more variety.

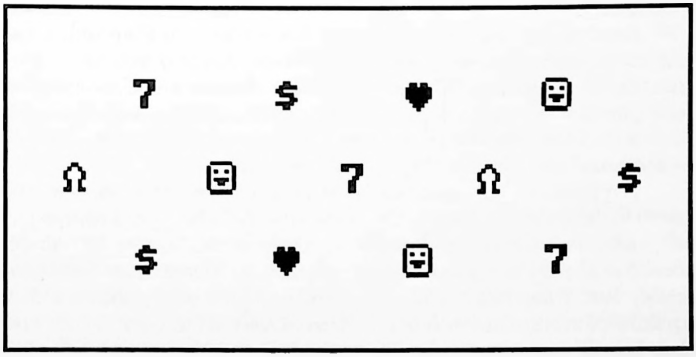
10. Expand the game to more than four keys. Use the Function keys instead. Let the game evolve into successive stages where first four keys are used, then six, then eight, and finally all ten.

With up to ten keys, the random tune played could be replaced by a short phrase from an actual piece of music. A table of various tunes could be coded from which selections could be made. At this point, the game has become a computer-based version of "Name that Tune."



2

"I Guess It's . . ."



Games in which you guess a number are often found in many computer gaming books. The overall thrust of most of these games can be summed up in the few lines of BASIC code that follow:

```
100 PRINT "I have a number between 0 and 999"
110 NUMB=INT(RND*1000)
120 INPUT "What is your guess? ",GUESS
130 IF GUESS<NUMB THEN PRINT "Too low" : GOTO 120
140 IF GUESS>NUMB THEN PRINT "Too high": GOTO 120
150 PRINT "Just right.  You got it!!"
```

Line 100 introduces the game and line 110 selects the random number that the player must guess. Lines 120 to 140 are repeated for each guess where the program checks if your guess is below or above the secret number; if it's neither, then you have guessed the correct number. Line 150 takes care of this with an "atta boy" and the game ends.

Some programs add variety to the game with such enhancements as using letters instead of numbers, limiting you to a fixed number of guesses, allowing two guesses at a time, or allowing only one guess in the event that you score points based on how close you are to the hidden number. Another variation is to change the clue to something other than a "high" or "low" message.

Suppose we had a numeric measure of how close our guesses were to the hidden number. This measure could then be displayed in a graphics format so that our visual acuity would have to be utilized in order to play the game. Our first game for this chapter does precisely that. Titled Hot 'n' Cold, it shows how good your guess is with a graphical thermometer. A good guess raises the mercury; a poor one has hardly any effect on the thermometer.

Hot 'n' Cold

The program Hot 'n' Cold functions as follows:

1. Pick a random number for the player to guess.
2. Display a thermometer using the PC's business graphics characters.
3. Have the player tell you his guess.
4. Calculate how close the guess is to the hidden number.
5. Display how close the player's guess is to the hidden number by letting the "mercury" in the thermometer rise to a point that indicates how close; then let the "mercury" fall back to the bottom of the thermometer.
6. If this guess is not the hidden number, repeat steps 3 through 5, or display an "atta boy" and end the game.

Step 1 is pretty simple. Multiply the random number function, RND, times the upper limit plus one of our secret number range. Convert the result to an integer and our secret number will be from zero to the upper limit. All this just means

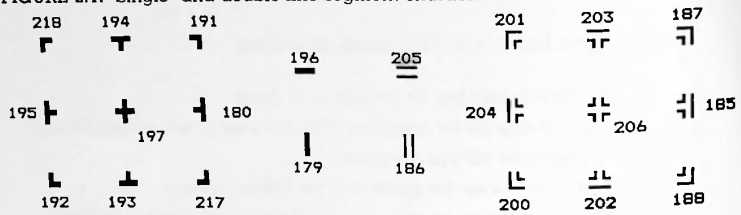
```
N = INT(RND*1000)
```

where the range is 0 to 999. We have to use the RANDOMIZE statement at the beginning of our program to insure different secret numbers are chosen by our program. Otherwise, every time we play, the game will begin with the same secret number.

A graphical thermometer can be designed using some of the business graphics characters on the PC. For example, the character codes 179 to 218 are single- and double-line segments. They can be printed in groups to display boxes, column separators, bar graphs, and the like. Some of them were used in our first program, PC Says, as a border around the players' scores. These same characters will be used extensively in most of our programs to depict what is going on graphically. Figure 2.1 organizes and illustrates the various codes that are used most often in our games.

A thermometer features a tube in which its mercury can fall and rise, with a numeric scale beside it. The tube is usually attached to a board that holds it erect. From a graphical standpoint, we will let the tube be composed of double-line-segment characters. These character codes are 201, 187, 200, and 188 for the corners, and codes 205 and 186 for the horizontal and vertical sides.

FIGURE 2.1. Single- and double-line-segment character codes.



The border surrounding the thermometer consists of single-line-segment characters. These character codes are 218, 191, 192, and 217 for the corners, and codes 196 and 179 for the horizontal and vertical sides.

A scale can be displayed by printing a numeric value along the side of a vertical double-line-segment character which also has a horizontal line segment. These character codes are 199 and 182, which will be alternatively printed with the vertical code 186, which has the horizontal line segment. It will provide the "tick" marks for our thermometer's scale.

The only thing left to consider is the size of the thermometer. The scale will be twenty character rows with the numbers 1 to 10 printed on every other row. Add another character space to everything for the border, and the thermometer can be neatly fitted vertically on the screen from rows 4 to 25.

The BASIC code to print the ideas developed above is listed in Program Segment 2.2.

```
190 FOR R=4 TO 22 STEP 2
200 COLOR 7 : LOCATE R,25 : PRINT USING"## ";10-((R-4)\2); ' scale
210 COLOR 6      : PRINT CHR$(199);"    ";CHR$(182);  ' sides with tic marks
220 LOCATE R+1,28: PRINT CHR$(186);"    ";CHR$(186);  ' sides  without 'em
230 NEXT
240 LOCATE 3,28:PRINT CHR$(201);STRING$(3,205);CHR$(187);  ' top thermometer
250 LOCATE 24,28:PRINT CHR$(200);STRING$(3,205);CHR$(188);  '  and bottom
```

Lines 190 to 230 print the vertical portion of the thermometer tube. The FOR / NEXT loop increments by two so that two PRINTs can be coded: one for the line with a scale value and one for the line without a scale. The R variable is the row where the first print takes place. The second print LOCATEs to row R + 1 for the unscaled line. Lines 240 and 250 print the top and bottom of the tube.

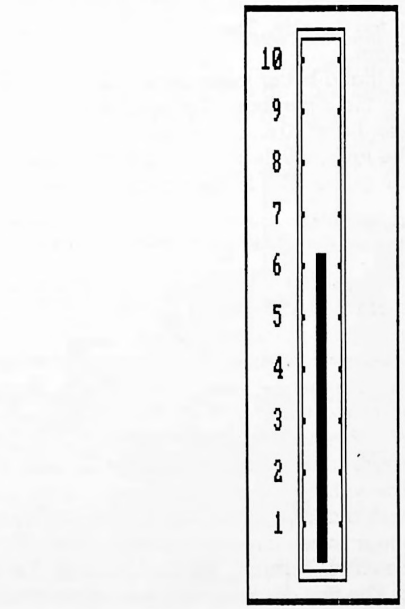
Printing the border that surrounds the tube is similar to the above but without the scale values.

The FOR / NEXT loop prints the two sides using lines 270 through 290 as shown in Program 2.3. Lines 300 and 310 print the four corners and the top and bottom edges and bottom lines of the border. See Figure 2.2 for an example of the thermometer and the game being played.

```
270 FOR R=3 TO 24
280 LOCATE R,23:PRINT CHR$(179);:LOCATE ,37:PRINT CHR$(179); ' border sides
290 NEXT
300 LOCATE 2,23:PRINT CHR$(218);STRING$(13,196);CHR$(191);    ' border top
310 LOCATE 25,23:PRINT CHR$(192);STRING$(13,196);CHR$(217);   '  and bottom
```

It's a simple matter to get the player's guess with a LINE INPUT statement. We use it instead of simply INPUT to filter out any non-numeric entries our players might make. The value from the LINE

FIGURE 2.2. Hotncold—Hot 'n' Cold guessing game.



INPUT is then converted to a numeric value via the VAL function before being used.

Calculating how close the guess is to the secret number can be done in a number of different ways. An initial approach might be to first determine the difference between the guess and the secret number. Then take the absolute value of the difference (that is, if the difference is less than zero, convert it to a positive number) and compare it to the overall range of secret numbers that is possible (1000 in our program).

This provides a percentage measure where, as the percentage becomes lower, it means the guess is closer to the secret number. There's a problem with this approach in that we want to convert our measure of closeness to a 1-to-10 scale (actually, 1-to-20 rows) on the thermometer.

When using a percentage value, the third or fourth guesses quickly fall within 75 to 85 percent of the guess. The thermometer shows a value of 8 or 9, and there's very little to be discerned as subsequent guesses are made. In other words, the closeness measure increases too quickly. The solution is complex in concept but simple to program.

It turns out that a better measure is to determine what power of two is closest to the difference. This number or exponent will be a number between 1 and 10 since 2 to the power of 10 is 1024, which is greater than the largest difference we can have between the secret number and the guess. The following code is used:

```
410 G=VAL(G$) : C=ABS(N-G)                    ' absolute guess difference
420 FOR S=1 TO 20 : IF C>2^(S/2) THEN NEXT    ' determine how close
```

First, the string value of the guess, G$, is converted to a numeric value, G. The absolute difference, C, between this guess and the secret number is then calculated. Since our thermometer's scale is measured over 20 rows, we use a FOR loop variable, S, to provide us with a value that ranges between 1 and 20. The guess difference is then compared to the value of 2 raised to the S/2 power, which is done with the ^, or exponentiation operator. The loop increments S until the difference, calculated as variable C, is no longer greater than 2^(S/2). The value of S at that point becomes our measure of "closeness." From a mathematical standpoint, this code converts the absolute closeness value, C, to its logarithm, base 2.

We now need to "turn on" the thermometer. Two FOR / NEXT loops are used: The first decrements a row value from the bottom of

the thermometer (at the bottom of the screen) to its top. A second loop does the reverse: A row value is incremented from the top row to the bottom. On the way up, a block character, 219, is printed to represent the mercury. Each print is also followed by another FOR / NEXT loop for a delay as the character is displayed. On the way down, a blank character erases the previously printed block characters. One last FOR / NEXT loop is also used for a pause as the mercury "hits the top" or before the downward loop begins. The program code for this is:

```
440 '
450 ' let thermometer rise and fall based on "closeness" of guess
460 '
470 FOR R=23 TO S+3 STEP -1
480 LOCATE R,30 : PRINT CHR$(219); ' rises
490 FOR D=1 TO 200 : NEXT D  ' delay on way up
500 NEXT R
510 IF N=G THEN 620          ' is guess right?
520 FOR D=1 TO 2000 : NEXT D ' pause at top
530 FOR R=S+3 TO 23
540 LOCATE R,30 : PRINT " "; ' falls
550 FOR D=1 TO 100 : NEXT D  ' delay on way down
560 NEXT R
```

The major portions of Hot 'n' Cold have now been described; the complete code is shown in Program 2.1. A few additional points should be mentioned: After the player's guess is inputted, it is validated to ensure that the guess is within the limits of our secret number range, 0 to 999. This is done in lines 380 and 390. If the guess is invalid, the program responds with a BEEP and the player must reenter the guess.

The game ends when a correct guess is made—this is checked in line 510. If the guess is correct, then the game wrap-up is done in lines 620 to 670: The player is told how well he did and is offered a chance to play again. The player is instructed to make another guess after any incorrect guess.

```
0 '  HOTNCOLD  Guess the number (with a graphic thermometer)
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  Computer chooses a random number from 0 to 999 which you must
50 '  guess. A thermometer displays a measure of how close the guess
60 '  is from the computer's secret number.
70 '
80 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
90 DEFINT A-Z : RANDOMIZE VAL(RIGHT$(TIME$,2))
100 PRINT"HOTNCOLD - Number guessing game" : PRINT
110 PRINT "In this game I will choose a number from 0 to 999"
120 PRINT "which you must try and guess.": PRINT
130 PRINT "Press <SPACE> to begin play";
```

```
140 WHILE INKEY$<>" " : WEND
150 CLS
160 '
170 '  display the thermometer, and borders around it
180 '
190 FOR R=4 TO 22 STEP 2
200 COLOR 7 : LOCATE R,25 : PRINT USING"## ";10-((R-4)\2); ' scale
210 COLOR 6      : PRINT CHR$(199);"   ";CHR$(182);  ' sides with tic marks
220 LOCATE R+1,28: PRINT CHR$(186);"   ";CHR$(186);  ' sides  without 'em
230 NEXT
240 LOCATE 3,28:PRINT CHR$(201);STRING$(3,205);CHR$(187);  ' top thermometer
250 LOCATE 24,28:PRINT CHR$(200);STRING$(3,205);CHR$(188); '  and bottom
260 COLOR 2
270 FOR R=3 TO 24
280 LOCATE R,23:PRINT CHR$(179);:LOCATE ,37:PRINT CHR$(179); ' border sides
290 NEXT
300 LOCATE 2,23:PRINT CHR$(218);STRING$(13,196);CHR$(191);    ' border top
310 LOCATE 25,23:PRINT CHR$(192);STRING$(13,196);CHR$(217);   '  and bottom
320 '
330 '  begin the game
340 '
350 N=INT(RND*1000): T=1: COLOR 7,0  ' pick the number, set count of guesses
360 LOCATE 1,18 : PRINT SPC(62);     ' erase player's last entry
370 LOCATE 1,18 : PRINT "Guess #";T;": "; ' prompt for guess
380 LINE INPUT G$ :IF G$>="0" AND G$<="999" AND LEN(G$)<4 THEN 410 ' edit guess
390 LOCATE 1,36 : PRINT"Whoops!!  0 to 999 only!" : BEEP
400 FOR D=1 TO 2000 : NEXT : GOTO 360       ' pause after erroneous entry
410 G=VAL(G$) : C=ABS(N-G)                  ' absolute guess difference
420 FOR S=1 TO 20 : IF C>2^(S/2) THEN NEXT ' determine how close
430 COLOR 4,0
440 '
450 ' let thermometer rise and fall based on "closeness" of guess
460 '
470 FOR R=23 TO S+3 STEP -1
480 LOCATE R,30 : PRINT CHR$(219); ' rises
490 FOR D=1 TO 200 : NEXT D  ' delay on way up
500 NEXT R
510 IF N=G THEN 620          ' is guess right?
520 FOR D=1 TO 2000 : NEXT D ' pause at top
530 FOR R=S+3 TO 23
540 LOCATE R,30 : PRINT " "; ' falls
550 FOR D=1 TO 100 : NEXT D  ' delay on way down
560 NEXT R
570 COLOR 7: LOCATE 1,1:PRINT USING"Last guess: ###";G;
580 T=T+1: GOTO 360 ' increment guess count, and get another
590 '
600 ' end game when guess is correct
610 '
620 COLOR 7 : LOCATE 3,1 : PRINT "Good going!  You got"
630 IF T=1 THEN PRINT "it in 1 guess!!!" ELSE PRINT "in" T "guesses."
640 PRINT : PRINT "Play again (Y or N)? ";
650 C$=INKEY$ : IF C$="" THEN 650
660 IF C$="N" OR C$="n" THEN C$=CHR$(27) : GOTO 9050
670 IF C$="Y" OR C$="y" THEN 150 ELSE 650
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
```

```
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 2.1 HOTNCOLD - Hot 'n Cold guessing game

Before we begin with our next game, let's examine some of the underlying concepts that the game Hot 'n' Cold embodies. Consider the potential range of secret numbers as being on a line or a long ruler, where each number is listed. The computer secretly marks one of the numbers that we in turn must find. The best strategy for playing the game is to keep dividing the range of our guesses in half so that we quickly zero in on the secret number. In "computerese," this is known as a *binary search* and is a very powerful procedure for searching among an ordered list of items. From a geometric standpoint, the number we are searching for is a point that represents a zero-dimensional space, and the line on which the point is hidden is a one-dimensional space. Suppose the dimensions of our game (geometrically speaking) were changed?

A number of interesting possibilities present themselves. Instead of picking a point on a line, let's pick a point in a two-dimensional space. Two numbers for the coordinates of the secret point would have to be identified. For each guess, the computer could respond with an indication of four directions: Your guess is above, below, to the left, or to the right of the secret point (or north, south, east, and west).

Instead of a single point, have two secret numbers that must be guessed. The problem then becomes finding a secret line segment. If the search space is also a line, then the computer might respond to the guessing process with some indication of whether the guessed line and the secret line overlap. There may be no overlap, and, also, the guess might be totally within the secret line. This amounts to searching for a one-dimensional item in a one-dimensional space.

The secret line segment could also be hidden in two-dimensional space. Now the guess might generate a response that indicates whether or not it crosses the secret line, and by what angle if it does. This game consists of searching for a one-dimensional item in a two-dimensional space.

Guessing games based on the above concepts are fairly abstract but do offer some educational benefits. The concept of points, line segments, and two-dimensional space are some fundamental concepts in mathematics. Exercises with them might be of some

worth to students. Our next game continues the theme described in the previous few paragraphs.

Boxit—Where's the box?

The game of Boxit consists of the computer's picking a random rectangular area that the player must find. In terms of the previous discussion, this is finding a two-dimensional object in a two-dimensional space. A player's guess consists of entering two coordinates: one for the upper-left corner and the other for the lower-right corner of a rectangle or box. Each coordinate is a row and column position on the screen, so a total of four numbers is entered.

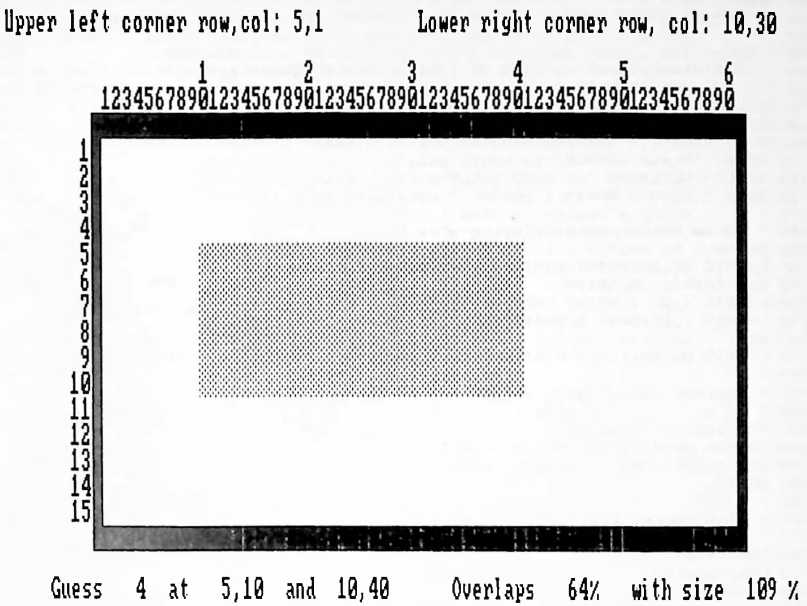
The computer responds with the following two pieces of information. First, the player's guess of the size of the hidden box and the actual size of the hidden box are compared. A percentage measure of how big the player's box is to the hidden box is reported. If the player's box is twice the size of the hidden box, then the measure reported is 200 percent; if it is half the size, then the measure reported is 50 percent, and so forth. Figure 2.3 illustrates the game being played.

Next, the amount of overlap between the two boxes is determined. This amount is also reported as a percentage. If the player's guess completely surrounds the hidden box, then the overlap is 100 percent; if there is no overlap, the percentage will be zero. This percentage will always be in the range of 0 to 100 percent; the size percentage may be well above 100 percent, depending on the size of the hidden box and the player's guess.

The PC lends itself nicely to a graphical representation of the game. Let the playing area, or the two-dimensional space, where the box is hidden be a large part of the screen. A border can be printed around the area using the same techniques employed by our PC Says and Hot 'n' Cold games. Each time a guess is made, a shaded area can be displayed that represents the player's guess. The computer's responses can be displayed below the border and the player guesses entered on the top line.

Program Boxit follows the same form of Hot 'n' Cold. The complete program code is listed in Program 2.2. First, the program starts up with the player defining the overall playing area. The variables BH and BW take on the number of rows, height, and number of

FIGURE 2.3. BOXIT—Where's the box?



columns, or width, in the playing area. A solid block character (code 219) border is printed around this area.

As an aid to the player in making his guesses, the program prints a list of row and column coordinates along the left side and top of the border. Headings for the computer's response to each guess are also printed at the bottom of the screen. Then the game is ready to begin. The four sections of code found in lines 130 to 410 perform these tasks, which essentially set up the screen for the game.

To pick a random box, the computer first chooses a set of upper-left corner coordinates. A random value is then added to these values to determine the lower-right corner. The calculations ensure that the point is within the coordinates for the playing area by basing the point selection on the playing-area size variables BH and BW. This is done in lines 420 to 490.

```
0 ' BOXIT  Guess coordinates of the box
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 ' Enter two number pairs to guess location of a random box.
50 ' (A 2 dimensional version of typical number guess games.)
60 '
70 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
80 DEFINT A-Z : RANDOMIZE VAL(RIGHT$(TIME$,2))
90 PRINT "BOXIT - Box coordinates Guessing Game" : PRINT
100 PRINT "Press <SPACE> to begin play"
110 WHILE INKEY$<>" " : WEND : CLS
120 BR=5 : BC=9 : BH=15 : BW=60  ' constants to define playing area
130 '
140 '  draw border about playing area
150 '
160 LOCATE BR,BC:PRINT STRING$(BW+2,219);  ' top
170 FOR I=BR+1 TO BR+BH
180 LOCATE I,BC : PRINT CHR$(219);          ' right side
190 LOCATE I,BC+BW+1 : PRINT CHR$(219);     ' left side
200 NEXT
210 LOCATE BR+BH+1,BC:PRINT STRING$(BW+2,219);  ' bottom
220 '
230 '  Column Coordinates across top
240 '
250 FOR I=BC+1 TO BC+BW
260 LOCATE BR-1,I : PRINT USING"#";(I-BC) MOD 10;
270 IF ((I-BC) MOD 10)=0 THEN LOCATE BR-2,I : PRINT USING"#";(I-BC)\10;
280 NEXT
290 '
300 '  Row Coordinates along side
310 '
320 FOR I=BR+1 TO BR+BH
330 LOCATE I,BC-2 : PRINT USING"##";(I-BR);
340 NEXT
350 '
360 '  Messages for last guess, and percentages
370 '
380 LOCATE 23,5 : PRINT"Guess";
390 LOCATE  ,16 : PRINT"at";
400 LOCATE  ,43 : PRINT"Overlaps     %";
410 LOCATE  ,60 : PRINT"with size      %";
420 '
430 ' pick random box coordinates for player to guess
440 '
450 CC1=INT(RND*BW)+BC+1              ' upper left corner column
460 CC2=INT(RND*(BW+BC-CC1))+CC1      '       and row
470 CR1=INT(RND*BH)+BR+1              ' lower right corner column
480 CR2=INT(RND*(BH+BR-CR1))+CR1      '       and row
490 CA=(CC2-CC1+1)*(CR2-CR1+1) : G=0  ' area of box, initialize guess count
500 '
510 '  begin play ... get players guess
520 '
530 LOCATE 1,1 : PRINT SPC(79);
540 LOCATE 1,1 : LINE INPUT; "Upper left corner row,col: ",G$
550 PR1=VAL(G$) : PC1=VAL(MID$(G$,INSTR(G$,",")+1))
560 IF PR1<1 OR PR1>BH OR PC1<1 OR PC1>BW THEN BEEP : GOTO 530   ' valid?
570 LOCATE 1,40 : PRINT SPC(39);
580 LOCATE 1,40 : LINE INPUT; "Lower right corner row, col: ",G$
```

```
590 PR2=VAL(G$) : PC2=VAL(MID$(G$,INSTR(G$,",")+1))
600 IF PR2<PR1 OR PR2>BH OR PC2<PC1 OR PC2>BW THEN BEEP : GOTO 570  ' valid?
610 '
620 ' display guess numbers, erase previous, and display new guess
630 '
640 G=G+1 : LOCATE 23,12 : PRINT USING "##" ; G ;
650 LOCATE 23,20 : PRINT USING "##_,##  and  ##_,##" ; PR1;PC1;PR2;PC2 ;
660 IF G>1 THEN CC=32 : GOSUB 930        ' erase previous guess
670 PR1=PR1+BR : PC1=PC1+BC : PR2=PR2+BR : PC2=PC2+BC    ' adjust coordinates
680 R1=PR1 : C1=PC1 : R2=PR2 : C2=PC2 : CC=176 : GOSUB 930 ' display new guess
690 '
700 ' determine overlap coordinates, player/overlap area, and %'s
710 '
720 PA=(PC2-PC1+1)*(PR2-PR1+1)              ' area of player's guess
730 MINR=PR1 : IF CR1>PR1 THEN MINR=CR1    '  upper left row
740 MINC=PC1 : IF CC1>PC1 THEN MINC=CC1    '          and column of overlap
750 MAXR=PR2 : IF CR2<PR2 THEN MAXR=CR2    '  lower right row
760 MAXC=PC2 : IF CC2<PC2 THEN MAXC=CC2    '          and column of overlap
770 IF MAXR<MINR OR MAXC<MINC THEN OA=0 : GOTO 790
780 OA=(ABS(MAXR-MINR)+1)*(ABS(MAXC-MINC)+1)   '  overlap area
790 PCOV!=OA/CA*100 : PSIZ!=PA/CA*100          '    and percentages
800 LOCATE 23,53 : PRINT USING "###" ; PCOV! ;
810 LOCATE 23,70 : PRINT USING "####" ; PSIZ! ;
820 '
830 ' player got it if both % are 100, otherwise guess again...
840 '
850 IF NOT (PCOV!=100 AND PSIZ!=100) THEN 530    ' continue if both not 100%
860 LOCATE 1,1 : PRINT SPC(79); : LOCATE 1,26    ' or end game if they are
870 PRINT "Good going.  You got it in ";G;
880 IF G=1 THEN PRINT "guess!!!" ELSE PRINT "guesses."
890 GOTO 9030
900 '
910 ' box filled with character CHR$(cc): erase with CC=32, fill with CC=176
920 '
930 FOR I=R1 TO R2
940 LOCATE I,C1 : PRINT STRING$(C2-C1+1,CC);
950 NEXT
960 RETURN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 2.2 BOXIT – Box coordinates guessing game

For each player's guess, the program ensures that each coordinate is within the playing area and that the second coordinate is greater than the first one. The player's previous guess is erased from the screen and the new guess, in the form of a shaded area, is displayed on the screen. Erasing the previous guess keeps the playing area from being overrun by the guesses as the game is played.

The subroutine in lines 900 to 960 performs both the erasure and shading by using the variable CC, which is set to a blank character code, 32, for erasing an area and to the shading character code, 176, to display the choice.

The percentage-size comparison consists of calculating the area of the player's guess, dividing it by the hidden box's area, and multiplying it by one hundred. The percentage of overlap is somewhat more involved.

The box represented by the player's guess may completely cover, partially cover, or be completely outside of the hidden box. To calculate the overlap area, we must first determine its coordinates. The upper-left corner of the overlap is the maximum of the player's upper-left corner point guess and the corresponding point for the hidden box. The lower-right corner of the overlap is the minimum of the player's guess and the lower-right corner of the hidden box.

Before using these point values to calculate the overlap area, there is one more check that must be made. If the lower-right corner's row or column point value is less than the corresponding point for the upper-left corner, then the overlap area is zero; that is, there is no overlap. These calculations are done in lines 690 to 790 of the program.

After calculating both percentages, the values are printed below the playing area and the game continues with the next guess. If both values are 100 percent, then the player has "boxed" the hidden box and the game is over. The number of guesses and the coordinates of the current guess are also displayed. This is so the player can keep track of what his last guess was.

An optimal playing strategy for BOXIT resembles searching for a point on a line. Focus in on the sides first, then the top and bottom. For a playing area of fifteen rows by sixty columns, fifteen to twenty guesses is a pretty good score.

We will close our chapter on guessing games with an old favorite that has been the basis for a number of television game shows and board games.

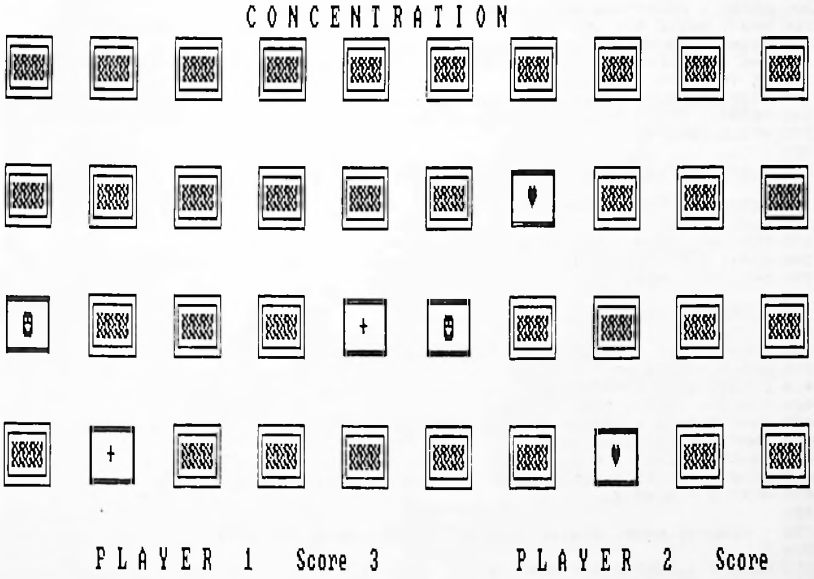
Concentration

The game of Concentration consists of picking pairs of objects from a limited group. If the two choices match, then the player gets another

choice; if the two choices do not match, the next player takes a turn in the same manner. The first player to match more than half the objects wins the game. Often, the objects being paired are playing cards placed face down in a rectangular arrangement of rows and columns. As they are chosen, the cards are turned over, face up, and, if paired, are removed. If they do not match as a pair, the cards are then turned back over and the next player takes a turn. See Figure 2.4 for an example of the game.

Each player selects a card by using the cursor keys to move the cursor to a point below the card. A card is selected by pressing the space key (or space bar), after which the card above the cursor is turned over. A second choice is then asked of the player. This processing of each choice occurs in the FOR / NEXT loop—which begins on line 660 in the program. The loop uses the variable CH to indicate which choice is being made, either 1 or 2.

FIGURE 2.4. Concentration game example.



Program 2.3, CONC, is a PC version of this game. The next few paragraphs will review some additional key points that make up the game and its program code.

```
0 '  CONC   Concentration - match pairs of cards
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  One or two players play trying to pair two cards.   Cursor
50 '  control keys move from card to card with a selection made by
60 '  pressing enter under the chosen card.
70 '
80 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
90 DEFINT A-Z : RANDOMIZE VAL(RIGHT$(TIME$,2))
100 DIM P(4,10), S(20) ' card pointers P, and symbols S
110 PRINT"CONC - Concentration" : PRINT
120 PRINT"Pick the card pairs.  Move cursor under"
130 PRINT"a card and press <SPACE> to select it."
140 PRINT"Match two cards and score a point." : PRINT
150 '
160 '  get number of players and playing area size
170 '
180 PRINT : PRINT"How many players (1 or 2)?";
190 NP=0 : WHILE NOT (NP=1 OR NP=2) : NP=VAL(INKEY$) : WEND : PRINT NP
200 PRINT : PRINT"How many rows of cards (2 to 4)? ";
210 H=0 : WHILE NOT (H>=2 AND H<=4) : H=VAL(INKEY$) : WEND : PRINT H
220 PRINT : PRINT"How many columns of cards (2 to 9)?";
230 W=0 : WHILE NOT (W>=2 AND W<=9) : W=VAL(INKEY$) : WEND : PRINT W
240 IF (H*W MOD 2)=0 THEN 260
250 PRINT : PRINT "This gives";H*W;"cards. I need an even number!" : GOTO 200
260 PRINT : PRINT "Press <SPACE> to begin game"
270 WHILE INKEY$<>" " : WEND
280 '
290 ' initialize playing card, P, and card symbol, S, array
300 '
310 RESTORE : FOR I=1 TO 20 : READ S(I) : NEXT
320 CLS : P=-1
330 FOR CR=1 TO H : FOR CC=1 TO W
340 P=P+1 : P(CR,CC)=S(P\2+1) : GOSUB 1220 ' init and display cards
350 NEXT CC : NEXT CR
360 '
370 ' shuffle cards
380 '
390 LOCATE 22,1 : PRINT"Press <SPACE> to shuffle cards.";
400 WHILE INKEY$<>" " : WEND
410 LOCATE 22,1 : PRINT SPC(35); ' clear shuffle message
420 FOR I=1 TO H : FOR J=1 TO W
430 R1=I : C1=J : R2=INT(RND*H)+1 : C2=INT(RND*W)+1
440 SWAP P(R1,C1),P(R2,C2)      ' SWAP two card values
450 CR=R1 : CC=C1 : GOSUB 1130  ' display first one
460 CR=R2 : CC=C2 : GOSUB 1130  '   then second one
470 NEXT J : NEXT I
480 '
490 ' display name, player scores, and init some variables
500 '
510 LOCATE 1,25:PRINT"C O N C E N T R A T I O N";
520 LOCATE 22,10:PRINT" P L A Y E R   1     Score";
530 LOCATE 22,50:PRINT" P L A Y E R   2     Score";
```

```
540 PS(1)=0 : PS(2)=0 : CR=1 : CC=1 : G=0 ' init scores, coords., guesses
550 SW=CINT(W*H/2) : IF NP=2 THEN SW=CINT(SW/2) ' score needed to win
560 '
570 ' begin game
580 '
590 FOR P=1 TO NP  '  for each player
600 LOCATE 22,(P-1)*40+10:COLOR 0,7:PRINT" P L A Y E R  ";P:COLOR 7,0
610 LOCATE 24,10:PRINT "Move to cards with cursor keys; ";
620 PRINT "Press <SPACE> to turn card over";
630 '
640 ' get two choices
650 '
660 FOR CH=1 TO 2  '  for each choice
670 LOCATE (CR-1)*5+5,(CC-1)*8+4,1
680 C$=INKEY$ : IF C$="" THEN 680      ' get a keystroke
690 IF C$=CHR$(32) THEN 800            ' SPACE pressed?
700 IF LEN(C$)=2 THEN C$=RIGHT$(C$,1) ' check for special key
710 ON INSTR("8246HPKM",C$) GOTO 730,740,750,760,730,740,750,760
720 BEEP : GOTO 680 ' invalid key pressed
730 IF CR>1 THEN CR=CR-1 : GOTO 670 ELSE 720 ' cursor up
740 IF CR<H THEN CR=CR+1 : GOTO 670 ELSE 720 ' cursor down
750 IF CC>1 THEN CC=CC-1 : GOTO 670 ELSE 720 ' cursor left
760 IF CC<W THEN CC=CC+1 : GOTO 670 ELSE 720 ' cursor right
770 '
780 ' SPACE pressed for a card not already selected?
790 '
800 IF P(CR,CC)>0 THEN IF CH=1 THEN 820 ELSE IF CR(1)<>CR OR CC(1)<>CC THEN 820
810 GOTO 720
820 GOSUB 1220 : CR(CH)=CR : CC(CH)=CC : PC(CH)=P(CR,CC)  ' display and save
830 NEXT CH  ' next choice
840 '
850 ' compare choices, continue same player if a match, else next player
860 '
870 G=G+1 ' up guess count
880 IF PC(1)=PC(2) THEN PS(P)=PS(P)+1 ELSE 950 ' do choices match?
890 LOCATE 22,(P-1)*40+36:PRINT USING"##";PS(P); : IF PS(P)>=SW THEN 1050
900 P(CR(1),CC(1))=0 : P(CR(2),CC(2))=0 ' reset card value array
910 GOTO 660  ' player continues
920 '
930 ' player missed; turn over cards, and continue with next player
940 '
950 LOCATE 24,10 : PRINT SPC(65); ' clear previous text
960 LOCATE 24,15 : PRINT"Whoops!  No match...  press <SPACE> to continue.";
970 WHILE INKEY$<>" " : WEND
980 CR=CR(1) : CC=CC(1) : GOSUB 1130  ' re-set first choice
990 CR=CR(2) : CC=CC(2) : GOSUB 1130  ' and the second one
1000 LOCATE 22,(P-1)*40+10:PRINT" P L A Y E R  ";P; ' clear player hi-light
1010 NEXT P : GOTO 590 ' continue play
1020 '
1030 ' game wrap-up: someone scored enough to win
1040 '
1050 LOCATE 24,10 : PRINT SPC(65); ' clear previous text
1060 LOCATE 24,20 : PRINT"Game over.  ";
1070 IF NP=1 THEN PRINT "You did it in";G;"guesses."; : GOTO 1090
1080 PRINT "Player";P;"wins";PS(P);"to";PS(1-(P=1));
1090 GOTO 9030 ' bypass subroutines
1100 '
1110 ' print back of card
1120 '
```

Program 2.3, CONC, is a PC version of this game. The next few paragraphs will review some additional key points that make up the game and its program code.

```
0 '  CONC   Concentration - match pairs of cards
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  One or two players play trying to pair two cards.   Cursor
50 '  control keys move from card to card with a selection made by
60 '  pressing enter under the chosen card.
70 '
80 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
90 DEFINT A-Z : RANDOMIZE VAL(RIGHT$(TIME$,2))
100 DIM P(4,10), S(20) ' card pointers P, and symbols S
110 PRINT"CONC - Concentration" : PRINT
120 PRINT"Pick the card pairs.  Move cursor under"
130 PRINT"a card and press <SPACE> to select it."
140 PRINT"Match two cards and score a point." : PRINT
150 '
160 '  get number of players and playing area size
170 '
180 PRINT : PRINT"How many players (1 or 2)?";
190 NP=0 : WHILE NOT (NP=1 OR NP=2) : NP=VAL(INKEY$) : WEND : PRINT NP
200 PRINT : PRINT"How many rows of cards (2 to 4)? ";
210 H=0 : WHILE NOT (H>=2 AND H<=4) : H=VAL(INKEY$) : WEND : PRINT H
220 PRINT : PRINT"How many columns of cards (2 to 9)?";
230 W=0 : WHILE NOT (W>=2 AND W<=9) : W=VAL(INKEY$) : WEND : PRINT W
240 IF (H*W MOD 2)=0 THEN 260
250 PRINT : PRINT "This gives";H*W;"cards. I need an even number!" : GOTO 200
260 PRINT : PRINT "Press <SPACE> to begin game"
270 WHILE INKEY$<>" " : WEND
280 '
290 ' initialize playing card, P, and card symbol, S, array
300 '
310 RESTORE : FOR I=1 TO 20 : READ S(I) : NEXT
320 CLS : P=-1
330 FOR CR=1 TO H : FOR CC=1 TO W
340 P=P+1 : P(CR,CC)=S(P\2+1) : GOSUB 1220 ' init and display cards
350 NEXT CC : NEXT CR
360 '
370 ' shuffle cards
380 '
390 LOCATE 22,1 : PRINT"Press <SPACE> to shuffle cards.";
400 WHILE INKEY$<>" " : WEND
410 LOCATE 22,1 : PRINT SPC(35); ' clear shuffle message
420 FOR I=1 TO H : FOR J=1 TO W
430 R1=I : C1=J : R2=INT(RND*H)+1 : C2=INT(RND*W)+1
440 SWAP P(R1,C1),P(R2,C2)      ' SWAP two card values
450 CR=R1 : CC=C1 : GOSUB 1130   ' display first one
460 CR=R2 : CC=C2 : GOSUB 1130   '   then second one
470 NEXT J : NEXT I
480 '
490 ' display name, player scores, and init some variables
500 '
510 LOCATE 1,25:PRINT"C O N C E N T R A T I O N";
520 LOCATE 22,10:PRINT" P L A Y E R   1    Score";
530 LOCATE 22,50:PRINT" P L A Y E R   2    Score";
```

```
540 PS(1)=0 : PS(2)=0 : CR=1 : CC=1 : G=0 ' init scores, coords., guesses
550 SW=CINT(W*H/2) : IF NP=2 THEN SW=CINT(SW/2) ' score needed to win
560 '
570 ' begin game
580 '
590 FOR P=1 TO NP  '  for each player
600 LOCATE 22,(P-1)*40+10:COLOR 0,7:PRINT" P L A Y E R  ";P:COLOR 7,0
610 LOCATE 24,10:PRINT "Move to cards with cursor keys: ";
620 PRINT "Press <SPACE> to turn card over";
630 '
640 ' get two choices
650 '
660 FOR CH=1 TO 2  '  for each choice
670 LOCATE (CR-1)*5+5,(CC-1)*8+4,1
680 C$=INKEY$ : IF C$="" THEN 680      ' get a keystroke
690 IF C$=CHR$(32) THEN 800            ' SPACE pressed?
700 IF LEN(C$)=2 THEN C$=RIGHT$(C$,1) ' check for special key
710 ON INSTR("8246HPKM",C$) GOTO 730,740,750,760,730,740,750,760
720 BEEP : GOTO 680 ' invalid key pressed
730 IF CR>1 THEN CR=CR-1 : GOTO 670 ELSE 720 ' cursor up
740 IF CR<H THEN CR=CR+1 : GOTO 670 ELSE 720 ' cursor down
750 IF CC>1 THEN CC=CC-1 : GOTO 670 ELSE 720 ' cursor left
760 IF CC<W THEN CC=CC+1 : GOTO 670 ELSE 720 ' cursor right
770 '
780 ' SPACE pressed for a card not already selected?
790 '
800 IF P(CR,CC)>0 THEN IF CH=1 THEN 820 ELSE IF CR(1)<>CR OR CC(1)<>CC THEN 820
810 GOTO 720
820 GOSUB 1220 : CR(CH)=CR : CC(CH)=CC : PC(CH)=P(CR,CC)  ' display and save
830 NEXT CH  ' next choice
840 '
850 ' compare choices, continue same player if a match, else next player
860 '
870 G=G+1 ' up guess count
880 IF PC(1)=PC(2) THEN PS(P)=PS(P)+1 ELSE 950 ' do choices match?
890 LOCATE 22,(P-1)*40+36:PRINT USING"##";PS(P); : IF PS(P)>=SW THEN 1050
900 P(CR(1),CC(1))=0 : P(CR(2),CC(2))=0 ' reset card value array
910 GOTO 660  ' player continues
920 '
930 ' player missed; turn over cards, and continue with next player
940 '
950 LOCATE 24,10 : PRINT SPC(65); ' clear previous text
960 LOCATE 24,15 : PRINT"Whoops!  No match...  press <SPACE> to continue.";
970 WHILE INKEY$<>" " : WEND
980 CR=CR(1) : CC=CC(1) : GOSUB 1130  ' re-set first choice
990 CR=CR(2) : CC=CC(2) : GOSUB 1130  ' and the second one
1000 LOCATE 22,(P-1)*40+10:PRINT" P L A Y E R  ";P; ' clear player hi-light
1010 NEXT P : GOTO 590 ' continue play
1020 '
1030 ' game wrap-up: someone scored enough to win
1040 '
1050 LOCATE 24,10 : PRINT SPC(65); ' clear previous text
1060 LOCATE 24,20 : PRINT"Game over.  ";
1070 IF NP=1 THEN PRINT "You did it in";G;"guesses."; : GOTO 1090
1080 PRINT "Player";P;"wins";PS(P);"to";PS(1-(P=1));
1090 GOTO 9030 ' bypass subroutines
1100 '
1110 ' print back of card
1120 '
```

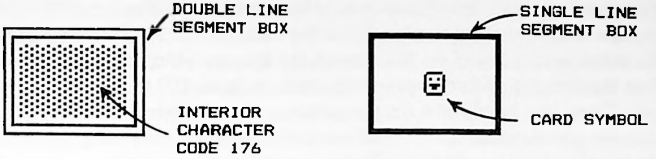
```
1130 COLOR 3 : R=(CR-1)*5+2 : C=(CC-1)*8+2
1140 LOCATE R+0,C : PRINT CHR$(201);STRING$(3,205);CHR$(187);
1150 LOCATE R+1,C : PRINT CHR$(186);STRING$(3,28);CHR$(186);
1160 LOCATE R+2,C : PRINT CHR$(200);STRING$(3,205);CHR$(188);
1170 COLOR 4 : LOCATE R+1,C+1 : PRINT STRING$(3,178);
1180 COLOR 7 : RETURN
1190 '
1200 ' print blank front of card
1210 '
1220 COLOR 5 : R=(CR-1)*5+2 : C=(CC-1)*8+2
1230 LOCATE R+0,C : PRINT CHR$(218);STRING$(3,196);CHR$(191);
1240 LOCATE R+1,C : PRINT CHR$(179);SPACE$(3);CHR$(179);
1250 LOCATE R+2,C : PRINT CHR$(192);STRING$(3,196);CHR$(217);
1260 COLOR 7 : LOCATE R+1,C+2 : PRINT CHR$(P(CR,CC))
1270 RETURN
1280 '
1290 ' symbols to use within cards
1300 '
1310 DATA 1,2,3,4,5,6,36,234,156,88,127,65,75,81,74,42,43,233,232,228,224
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 2.3 CONC - Concentration guessing game

The game is structured as follows. A set of playing cards are created with single-line segment boxes containing some unique PC graphics characters, such as the smiley face, the four card suit symbols, and so forth. The graphics character is placed in the middle of the box to represent the card's face. For the back of the card, the interior of the box is filled with some "shading" characters, either 176, 177, or 178. A box of three rows by five columns is used for the cards: This is small enough to allow up to forty cards on the screen yet large enough so that the cards can be easily distinguished from each other. Figure 2.5 illustrates the form that the individual cards take.

To provide some variety in the game, we can let the player decide how many cards will be used. Based on a three-row by five-column card size, a playing area of from two by two to four by nine cards will be allowed. This includes three spaces between each card column and two lines between each row. The player enters the playing area size by indicating the number of rows and columns of cards that will be used. Note that the product of these two values is the total number of cards and must be an even number since the cards must be in pairs. The variables H and W define the size of the playing

FIGURE 2.5. The front and back of a Concentration game card.



area, where H is the height or number of rows and W is the width or number of columns.

The playing card values are maintained in the two-dimensional array P, which is dimensioned at P(H,W), one entry for each card. Thus, P(1,1) is the upper-left corner card, P(1,2) is the one to its right, P(2,1) is the one below it, and so on. These card values must now be mapped into specific locations on the screen.

Let the row and column screen coordinate 2,2 be the upper-left corner of the card in P(1,1). To determine the card's position, multiply the card number row by 5, the card height of 3 plus 2 for spacing, and the card number column by 8, its width of 5 plus the 3 spaces. Since we want the first card at row 2, column 2, 1 is subtracted from the card number row and 2 is added to each product.

Thus, the card for P(1,1) is positioned at screen row and column 2,2, card P(2,2) is at 7,10, and so forth. The program uses the variables CR and CC to index the P array for card values, and converts them to their actual screen coordinates with:

$$R=(CR-1)*5+2 : C=(CC-1)*8+2$$

This code begins each of the two-card display subroutines located in lines 1100 to 1180 for the back of the card, and lines 1190 to 1270 for the front of the card.

The actual card values in the P array are read from a DATA statement. The values consist of many of the gaming symbols, character numbers 1 to 6, plus some other characters (such as the dollar sign), and some Greek character symbols for the entire group of 20. At least eighteen numbers are required since a maximum of 38 cards is possible.

The numbers are first loaded into the S array and then inserted in pairs into the P (playing card) array. An index to the card symbols in the S array is incremented by 1 but is divided by 2 when used to set

the card value in the array P. Hence, the resulting index value only increases every other time it is referenced, and the card symbol numbers are neatly set as pairs in the card array. At the same time, the card is displayed on the screen by the card-display subroutine. This is all done with the program code at lines 280 to 350.

Once the cards are on the screen, the program pauses so the players can familiarize themselves with the cards and their symbols. The cards are then shuffled: This is a matter of looping through each card in the array and exchanging it with some other randomly chosen card. At the same time, the two cards are "turned over" by printing the back of the card over its face. This part of the program can be found in lines 360 to 470.

A few other messages are added to the screen for the player numbers and scores. The name of the game is also printed at the top of the screen. Some additional variables are set such as the player score array's initial values, array PS, the initial location of the cursor, CR and CC, and the score needed to win, variable SW. Note that the score needed to win depends on whether there are one or two players. With the screen display finally completed, play can begin.

The actual input and output processing consists of the now-familiar C$=INKEY$ loop at line 680. The program checks for either a cursor control character, numeric keypad equivalent, or the space key. If a space is entered, then the program must also check that the card selected is not one that has already been matched or the same one that was selected with the player's first choice. The player's choices are saved in the two-element array PC. The position of each choice must also be saved—this is done in the arrays CR and CC and is needed to turn the cards back over if the cards do not match.

Processing of the cursor control keys consists of incrementing or decrementing the appropriate card position variable. The variable CC is the current column position and is incremented or decremented whenever a right or left cursor key is pressed, respectively. This is also done for the row–card position variable CR when the down or up keys are pressed. (Note that to keep from inadvertently moving beyond the bounds of the playing area, the CR and CC variables are also compared with 1 and the H and W the playing area height and width variables.) The code for this processing of the cursor keys is located at lines 720 to 760.

Line 670 converts the current card position, CR and CC, to a screen row and column position in the middle and just below the

current card. The cursor continues to blink at this position with a LOCATE statement so the player knows where his choices are being made.

After the two choices are indicated, they must be compared. If the choices match, the player scores a point and gets to make two more choices. If the choices do not match, a "Whoops. . ." message is displayed and the program pauses until the space bar is pressed. This pause is necessary because, without it, there would be no time to see the two cards just turned over. When a key is pressed, the positions for each choice are pulled from the CR and CC arrays and each card is turned over. The next player then begins play.

The program highlights the appropriate player number in the screen status area to indicate whose turn it is. After a player is finished, the player number is again printed, but in normal video to turn off the highlighting. This is done in lines 600 and 1000, respectively.

In addition to increasing the player's score when a match is made, the program also checks whether or not this player has matched enough of the cards to win. For example, if a player has tallied enough points to win based on the SW variable, the program concludes the game and a message to that effect is displayed. For solitaire games, the score to win is set at the total number of pairs. Here the game's challenge is to make the matches in as few guesses as possible.

Variations

As with all games, a number of variations can be easily considered. The guessing games in this chapter are no exception.

1. Change the program Hot 'n Cold so the player must find a hidden line segment. Let the thermometer represent the extent of overlap instead of closeness.

2. Convert Hot 'n Cold to a two-player game. Each player is guessing his own secret number. Put the thermometer display code in a subroutine and call it after each player's guess.

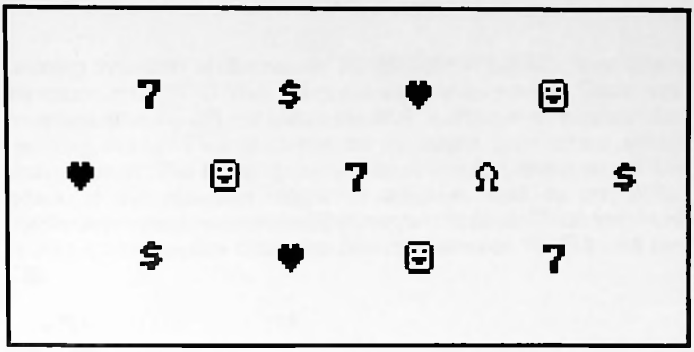
3. Add color to the program Boxit. As each choice is made, select another color for the shading. Thus, the player's previous choices can be kept on the screen.

4. Use a hidden line segment instead of a box in Boxit. Respond to the player's guess with an indication of whether the player's line guess intersects the hidden line and, if so, at what angle the intersection is.

5. Use a single point instead of a box in Boxit. Respond to each player's guess with a north, south, east, or west message.

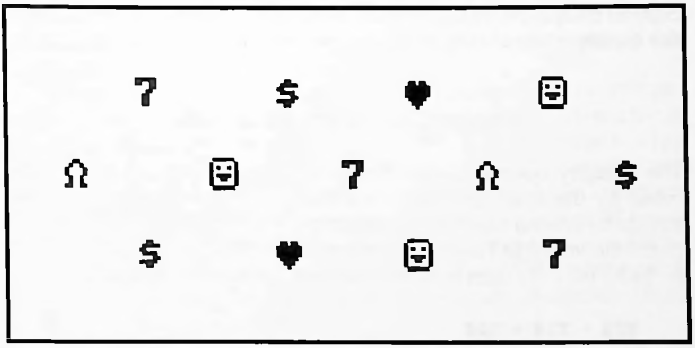
6. Add some wild cards (the smiley face, for example) that match any other card in Concentration. Note that this complicates the choice comparisons that must be made.

7. Assorted funny messages can be added to all of the programs to tell the players of the games how well or badly they are doing.



3

The Word Is . . .



Word games differ from typical computer games in a number of ways. First, programming word games involves manipulating character text or character strings as opposed to most games, which generally require processing integer numbers. Second, the concept of a word game can vary quite a bit, from a simple puzzle to be solved to vocabulary testing and sentence generation. Last, the need and use of graphics in word games is fairly limited. Character text within the puzzle or game is displayed in a fairly straight-forward manner on the screen. There is no need to design complex screen layouts, but of course, their inclusion in such games can liven up the play.

For our purposes, word games consist of a set of four programs that illustrate a number of techniques for manipulating character text for words, phrases, and complete sentences. Many people have difficulty in coding programs that manipulate character text. In general, the concepts are similar to those used in arithmetic processing.

PROCESSING CHARACTER DATA

The primary operation with strings is *concatenation*, which is indicated by the plus sign, as in addition. This operation creates new strings by linking together or joining two (or more) strings. For example, if the variable T1$ is the string of characters "TO BE OR" and T2$ is "NOT TO BE", then a third string can be formed as follows:

```
T3$ = T1$ + T2$
```

creating the string of characters "TO BE ORNOT TO BE". So where's the space in ORNOT? Well, what you see is what you get! There was no space after the OR nor before the NOT, so there is no space in the resulting string. This illustrates an important point about string manipulation: The blank space is just as much a character as the letters of the alphabet, digits, punctuation, and so on. Withoutblankspacesasentencewouldbeprettyhardtoread. Thus, we need to add a blank-space character constant between the T1$ and the T2$:

```
T3$ = T1$ + "  " + T2$
```

Anything in quotes is considered a string by BASIC. This differs from numbers that stand alone. Thus you cannot set the string variable A$ to the three characters XYZ as:

```
A$ = XYZ
```

You must instead code them as:

```
A$ = "XYZ"
```

Manipulating strings are done primarily with the following functions. The MID$ function allows portions of one string to be extracted from another. The LEFT$ and RIGHT$ functions are special cases of MID$: They are limited to extracting strings from the left or right, respectively, of their strings. Another useful string function is INSTR. It calculates a number value indicating where in one string you can find another string. (This function would be helpful in determining the position of "TO BE" in the earlier examples.)

Two other functions are often used with single character strings. Since computers can only work with numbers, every character is assigned a numeric value. Your PC and PCjr characters have numeric values based on what is known as the *ASCII character set,* which is a standard used by most computers. The BASIC function ASC returns the ASCII code for a character; for instance, if X$ has a value that is the character A, then ASC(X$) will return the number 65 to you. There is a converse to this function, the CHR$ function, that takes a number and converts it to the appropriate character. Thus, CHR$(65) is the character A.

One last comment should be made about character strings. There is a special type of character string known as the *null* string. It is a character string that has a length of zero, or a string containing no characters. Do not confuse this with a string of blank spaces (which may not appear on the screen as anything but still occupies a number of character positions in the string and on the screen—in fact, the blank space has an ASCII code of 32).

During string manipulations you might think of the null string as equivalent to the number zero. For instance, zero added to any number is that number. Likewise, concatenating the null string to any string is that string with no change. The null string is often used to initialize a string (as you often do when initializing numeric variables to zero) before manipulating a group of strings.

Four programs are developed in this chapter. First, there is RSEN for random sentences, which illustrates how words can be manipulated to create pseudo-English grammar. Another form of RSEN, but with "less random" sentences, can be created based on the user's entering text interactively. The program that does this is called ELSY. ELSY is a take-off on Eliza, the famous "conversational" program developed by Joseph Weizenbaum and his team of artificial intelligence researchers at MIT. Next, there is WPUZ, which illustrates a techique for generating word-find puzzles. Last, we have HMAN, the traditional word game of hangman.

RSEN—A Random Sentence Generator

Put a few random words together and—behold—you have a sentence. But it's not that simple when you consider the rigors of English grammar. There's also the not-so-simple problem of ensuring that the sentence makes logical sense. Before we consider sentences, let's look at phrases.

Consider the problem of creating buzzwords or high-tech phrases. The following table has three columns of computer-related terms:

1. on-line	information	system
2. real-time	communications	applications
3. quantitative	input/output	package

4. advanced	operations	process
5. data base	computing	software
6. decision	support	hardware

Randomly select a word from each column, put the three words together, and you have phrases such as "real-time operations hardware." It is quite easy to develop a BASIC program to do this. Let the three arrays WC1$, WC2$, and WC3$ correspond to the words in each column numbered from 1 to 6, such as WC1$(1) for "on-line" and WC3$(6) for "hardware." The program to generate the phrases is as follows:

```
100 K1=INT(RND*NW1)+1   ' set a random index for
110 K2=INT(RND*NW2)+1   '     each of the three
120 K3=INT(RND*NW3)+1   '     word arrays
130 WORD1$=WC1$(K1)     ' get a word from each of
140 WORD2$=WC2$(K2)     '     of the arrays using
150 WORD3$=WC3$(K3)     '     the random indices
160 BUZZ$=WORD1$+" "+WORD2$+" "+WORD3$   ' build a phrase
170 PRINT BUZZ$                          ' and display it
```

The arrays can be loaded from tables stored in the program as DATA statements. Now let's modify the above approach to create some complete sentences.

The thrust of our random sentence program is as follows. First, we will set up a number of distinct word arrays (such as the "columns" above). Then, we will randomly select words from the arrays. Our sentences will follow the pattern:

ARTICLE ADJECTIVE SUBJECT-NOUN PREP-PHRASE
ADVERB VERB ADJECTIVE OBJECT-NOUN PREP-PHRASE.

The program contains six string arrays named: SNOUN$, ADJ$, VERB$, ADVERB$, PREP$, and ONOUN$. Each array contains a set of subject nouns, adjectives, verbs, adverbs, prepositions, and object nouns. The sets are initialized from DATA statement tables located at the end of the program.

Once stored in the arrays, the words can be selected at random to create a sentence that corresponds to the pattern shown above. The prepositional phrase (PREP-PHRASE) will be handled in a special subroutine. It will be formed by using a random preposition followed by an article, adjective, and either a subject or an object noun.

PREP-PHRASE = PREPOSITION ARTICLE NOUN

To add further randomness to the program's sentences, certain parts of the sentence may be left out. In particular, a random number test precedes the selection of the adjectives so that the adjectives appear in the sentence only 50 percent of the time. Furthermore, the two adverbs are in the sentence only 50 percent of the time and prepositional phrases are included 30 percent of the time. These random proportions are obtained by using the RND function in an IF statement such as:

IF RND<.3. THEN — — — rest of line — — —

which processes the statement after the THEN statement approximately 30 percent of the time. Change the .3 to .5 to include the prepositional phrases 50 percent of the time. IF/THEN statements as shown above determine the random placement of items in the sentences generated by the program. Note that the minimum sentence will take the form:

ARTICLE SUBJECT-NOUN VERB OBJECT-NOUN.

A separate subroutine is used to tack on the articles "a," "an," and "the." This is done to ensure that "an" precedes words that begin with vowels. Figure 3.1 illustrates a few sentences generated by the program. Program 3.1, RSEN, is the complete random sentence generator.

To run the program, just press the space key and a new sentence will be created and displayed. The tables are dominated by computer-related terms, as in the buzzword generator. The tables can be easily modified to include additional words (just add the words to the appropriate table and change the number count). To exit from the program, press the Esc key.

ELSY—An "Eliza-like" Program

Several years ago, a program titled "Eliza" was developed to investigate the problem of extracting the meaning from free-form English text. The program has become famous for the seemingly "intelligent"

FIGURE 3.1. Some random sentences generated by RSEN.

```
RSEN  -  Random Sentence Generator

Press <SPACE> for a sentence,
or Esc to end the program.


A record runs ASCII equipment.


A disk quickly reports rapid equipment.


The printer executes slow network.


A PASCAL program slowly runs rapid network.


The software slowly operates information on a processor.


The byte quickly processes ASCII hardware.
```

```
0 '  RSEN   Random sentence generator
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  Randomly select words to construct a sentence.  Overall pattern is:
50 '
60 '    ARTICLE  ADJECTIVE  SUBJECT-NOUN  PREP-PHRASE
70 '                    ADVERB  VERB  ADJECTIVE  OBJECT-NOUN  PREP-PHRASE.
80 '
90 '  All but article, noun and verb may or may not be in sentence.
100 '
110 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
120 DEFINT A-Z : RANDOMIZE VAL(RIGHT$(TIME$,2)+MID$(TIME$,4,2))
130 PRINT"RSEN  -  Random Sentence Generator" : PRINT
140 '
150 '  load word table arrays
160 '
170 READ N1 : DIM SNOUN$(N1): FOR I=0 TO N1-1 : READ SNOUN$(I): NEXT
180 READ N2 : DIM ADJ$(N2)  : FOR I=0 TO N2-1 : READ ADJ$(I)  : NEXT
190 READ N3 : DIM VERB$(N3) : FOR I=0 TO N3-1 : READ VERB$(I) : NEXT
200 READ N4 : DIM ADV$(N4)  : FOR I=0 TO N4-1 : READ ADV$(I)  : NEXT
210 READ N5 : DIM PREP$(N5) : FOR I=0 TO N5-1 : READ PREP$(I) : NEXT
220 READ N6 : DIM ONOUN$(N6): FOR I=0 TO N6-1 : READ ONOUN$(I): NEXT
230 '
240 '  now do it...
250 '
260 PRINT"Press <SPACE> for a sentence,"
270 PRINT"or Esc to end the program." : PRINT
280 C$="" : WHILE C$="" : C$=INKEY$ : WEND
290 IF ASC(C$)=27 THEN 9050 ELSE IF C$<>" " THEN 280
300 GOSUB 340 : PRINT : PRINT S$+"." : PRINT : GOTO 280
310 '
```

```
320 ' build a sentence
330 '
340 W=INT(RND*N1) : S$=SNOUN$(W)                          ' start with a subject
350 IF RND>.5 THEN W=INT(RND*N2) : S$=ADJ$(W)+" "+S$      ' an adjective (maybe)
360 W$=S$ : GOSUB 710 : S$=A$+" "+S$                      ' pick an article
370 S$=CHR$(ASC(S$)-32)+MID$(S$,2)                        ' capitalize 1st letter
380 IF RND>.7 THEN GOSUB 770 : S$=S$+" "+P$               ' prepositional phrase
390 IF RND>.5 THEN W=INT(RND*N4) : S$=S$+" "+ADV$(W)      ' an adverb (maybe)
400 W=INT(RND*N3) : S$=S$+" "+VERB$(W)                    ' add the verb
410 IF RND>.5 THEN W=INT(RND*N2) : S$=S$+" "+ADJ$(W)      ' another adjective
420 W=INT(RND*N6) : S$=S$+" "+ONOUN$(W)                   ' subject-noun
430 IF RND>.7 THEN GOSUB 770 : S$=S$+" "+P$               ' another prep phrase
440 RETURN
450 '
460 ' word tables follow
470 '
480 DATA 12  : REM  subject nouns
490 DATA program, language, computer, data, byte, file, record
500 DATA software, printer, disk, monitor, processor
510 '
520 DATA 10 : REM  Adjectives
530 DATA BASIC, PASCAL, COBOL, large, small, fast, quick, slow, rapid, ASCII
540 '
550 DATA 10 : REM  Verbs
560 DATA communicates, accesses, retrieves, reports, processes, operates
570 DATA inputs, outputs, executes, runs
580 '
590 DATA 3  : REM  Adverbs
600 DATA  quickly, slowly, rapidly
610 '
620 DATA 9  : REM  Prepositions
630 DATA in, on, of, at, by, for, within, between, around
640 '
650 DATA 9  : REM  Object nouns
660 DATA hardware, system, machine, information, data-base, device
670 DATA network, data, equipment
680 '
690 '  select an article: a, an or the
700 '
710 IF RND<.5 THEN A$="the" : RETURN
720 IF INSTR("aeiou",LEFT$(W$,1))>0 THEN A$="an" ELSE A$="a"
730 RETURN
740 '
750 ' create a prepositional phrase (don't allow duplicate nouns)
760 '
770 W=INT(RND*N5) : P$=PREP$(W)
780 W=INT(RND*N1) : W$=SNOUN$(W) : IF INSTR(S$,W$)=0 THEN 800
790 W=INT(RND*N6) : W$=ONOUN$(W) : IF INSTR(S$,W$)<>0 THEN 790
800 GOSUB 710 : P$=P$+" "+A$+" "+W$
810 RETURN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 3.1 RSEN - Random sentence generator

dialogue it carried on with its user. Numerous versions of Eliza have been developed. A number of them present the dialogue as a therapy session with a psychiatrist. The computer's reply is often in the form of a leading question containing a word extracted from the user's input.

Our program, ELSY, is a take-off on this theme, in a somewhat simplified manner. It functions as follows:

1. Load some string arrays with random sentences and keywords from tables in the program.
2. Begin the conversation with a random statement.
3. Accept the user's reply.
4. Search through the reply, extracting all words (any group of characters preceded by a blank on either side).
5. Save groups of words (phrases) found in step 4 that are delimited by any of the keywords loaded in step 1 in an array.
6. After extracting all phrases in the user's reply, create the computer's response by either filling in the blanks of a random sentence with a random phrase from step 5 or with some miscellaneous sentence if no phrases were found.
7. Continue the conversation with step 3.

The user's reply is accepted via a LINE INPUT statement. Next, the user's reply is examined character by character using the MID$ and ASC functions, converting all upper-case letters to lower case. Then the program extracts each word in the user's reply by once again examining each character in the reply string. Each word extracted is stored in variable W$. Otherwise the word is discarded. Each time a blank is found, W$ is also checked; if a character is not null (that is, if it has one or more characters in it), then it is the word that preceded the blank we just found.

Once a word is found, it is compared to each entry in a keyword table. This table consists of words loaded into the array KWD$ from DATA statements in the program.

If a word from the user's reply is not in the program's keyword array, it is saved in the string PH$; otherwise the word is discarded. Each time the program finds a key word in the user's reply, it checks if any words are in the variable PH$. If words *were* found, then PH$ is saved as a phrase from the user's reply in the array KWR$. The

variable TW is used to count and keep track of the total number of phrases found.

As an example, let the words "the," "a," and "is" be keywords that the program looks for. Then, in the sentence:

```
The sky is a dark blue.
```

the phrases "sky" and "dark blue" are extracted by the program.

Once all the phrases are extracted, one of the phrases is randomly selected and inserted into a random sentence with which the program responds. A DATA statement table of such sentences (there are twenty-five in the program) is loaded into the array REPLY$. Each sentence contains a dash (—) which indicates where a particular phrase found in the user's sentence is to be inserted.

Insertion is a matter of using the INSTR function to find the location of the dash in the sentence, and replacing it with the random phrase. All characters to the left of the dash are concatenated to the left of the phrase; those characters to the right of the dash are put on the right side of the phrase. The sentence is then displayed for the user.

To end your conversation with Elsy, type any sentence containing the words "bye" or "goodby" and the program will stop. A random closing remark is printed to conclude the conversation. Elsy always has the last word!

Using the program is simply a matter of keying in text at the greater-than sign (>). It takes but a moment for Elsy to process the entry and reply to it. The more unique the words in your sentence, the better the reply. Elsy is very attentive but not too discerning in what you say. See the sample conversation carried on in Figure 3.2. Program 3.2 is the complete program.

Many other approaches can be used in the program to create the replies. For example, the random reply sentences could be categorized and used based on specific keywords found in the user's reply. This would require quite a number of sentences for any meaningful set of keywords.

Another alternative might be to randomly create the computer's reply sentence, as in the previous Random Sentence program, RSEN. Also, keywords from previous user replies could be saved so that a degree of continuity could be achieved from sentence to sentence.

FIGURE 3.2. Conversing with Elsy.

```
ELSY  -  Eliza-like computer dialogue

Hello there!  I'm Elsy,  Eliza's cousin

Let's talk for a while.  Okay?

>I feel strange talking to a computer

Enough about computer; let's talk about you

>I rather talk about computers

What does rather talk about computers mean to you ?

>Computers should be seen and not heard

Computers is for you, not me

>But you are a computer

Computer must be important to you

>



You can say more than that!

>What about you

Does what about refer to me ?

>What about you

Hmmm... you just said that!

>bye

I enjoyed talking with you.
```

```
0 ' ELSY  Eliza-like interation with user
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  Program creates random sentences based on user inputs.
50 '
60 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
70 DEFINT A-Z : RANDOMIZE VAL(RIGHT$(TIME$,2)+MID$(TIME$,4,2))
80 PRINT "ELSY  -  Eliza-like computer dialogue" : PRINT : PRINT
90 PRINT "Hello there!  I'm Elsy,  Eliza's cousin" : PRINT
100 READ NS : DIM REPLY$(NS): FOR I=0 TO NS-1 : READ REPLY$(I): NEXT ' replies
110 READ KW : DIM KWD$(KW)  : FOR I=0 TO KW-1 : READ KWD$(I)  : NEXT ' keywords
120 DIM KWR$(50)  ' for words from user's reply
130 L$=""         ' last user input to check for repeats
```

```
140 '
150 ' start talking...
160 '
170 ON INT(RND*6)+1 GOSUB 760,770,780,790,800,810 ' opening statement
180 PRINT : LINE INPUT">",R$ : PRINT  '  get user's reply
190 IF R$="" THEN PRINT "You can say more than that!" : GOTO 180
200 IF R$=L$ THEN PRINT "Hmmm... you just said that!" : GOTO 180
210 L$=R$ : GOSUB 1020 '  extract words
220 FOR I=0 TO PH      '  check if user done
230 IF KWR$(I)="bye" OR KWR$(I)="goodbye" THEN 280 ' exit words
240 NEXT
250 IF TW>0 THEN GOSUB 330 : PRINT S$ : GOTO 180      ' create random reply, or
260 ON INT(RND*7)+1 GOSUB 850,860,870,880,890,900,910 ' use no words reply
270 GOTO 180
280 ON INT(RND*4)+1 GOSUB 950,960,970,980  ' closing remark
290 PRINT : PRINT : GOTO 9030
300 '
310 ' build a reply
320 '
330 KW$=KWR$(INT(RND*TW))  ' pick a keyword
340 S$=REPLY$(INT(NS*RND)) ' pick random reply
350 P=INSTR(S$,"-")        ' find dash (blank space)
360 S$=LEFT$(S$,P-1)+KW$+RIGHT$(S$,LEN(S$)-P)      ' replace it with keyword
370 IF S$>"Z" THEN S$=CHR$(ASC(S$)-32)+MID$(S$,2) ' capitalize 1st letter
380 RETURN
390 '
400 '  fill-in-the-blank reply sentences
410 '
420 DATA 25
430 DATA "Tell me more about -
440 DATA "- is very interesting
450 DATA "I do not understand -
460 DATA "Do you talk about - very often ?
470 DATA "What is - all about?
480 DATA "You can't say enough about - can you ?
490 DATA "Enough about -; let's talk about you
500 DATA "Is there more to - ?
510 DATA "Why do you mention - ?
520 DATA "Does - refer to me ?
530 DATA "Do you really believe in - ?
540 DATA "What about - ?
550 DATA "It's not really - is it ?
560 DATA "Does - bother you ?
570 DATA "What does - mean to you ?
580 DATA "- makes me wonder...
590 DATA "- is for you, not me
600 DATA "- cannot be what you really mean
610 DATA "- does nothing for me
620 DATA "- must be important to you
630 DATA "It's hard to say; - is very interesting
640 DATA "I feel - is a touchy subject
650 DATA "Many say that - is not the thing to say
660 DATA "Do you think - is very important ?
670 DATA "Something like - cannot be overlooked
680 '
690 DATA 40 : REM Keywords
700 DATA the, a, an, and, you, your, it's, is, are, was, in, on, of, so, like
710 DATA at, with, but, for, among, around, there, their, will, shall, do, be
```

```
720 DATA i, he, she, it, my, mine, his, her, can, may, should, could, did
730 '
740 ' opening remarks
750 '
760 PRINT"How are things with you today?":RETURN
770 PRINT"What would you like to talk about?":RETURN
780 PRINT"Care to chat for few moments?":RETURN
790 PRINT"So what's going on with you?":RETURN
800 PRINT"Let's talk for a while.  Okay?":RETURN
810 PRINT"Tell me about yourself...":RETURN
820 '
830 ' no key word remarks
840 '
850 PRINT "Oh... I understand.":RETURN
860 PRINT "What was that again?":RETURN
870 PRINT "Tell me more!":RETURN
880 PRINT "Why do you say that?":RETURN
890 PRINT "That's unusal, isn't it?":RETURN
900 PRINT "Hmmm... interesting.":RETURN
910 PRINT "Well! Is that so???":RETURN
920 '
930 ' closing remarks
940 '
950 PRINT "Come again some other time...":RETURN
960 PRINT "I enjoyed talking with you.":RETURN
970 PRINT "Good-bye for now.": RETURN
980 PRINT "So long... come again!":RETURN
990 '
1000 ' parse user's reply into words, check for keywords
1010 '
1020 W$="" : TW=-1 : PH$="" :  R$=R$+" "
1030 WHILE LEN(R$)>0
1040    C=ASC(R$) : R$=MID$(R$,2) : IF C>64 AND C<91 THEN C=C+32
1050    IF C=32 AND LEN(W$)>0 THEN GOSUB 1140:W$="" ' is it a keyword?
1060    IF C>32 AND C<128 THEN W$=W$+CHR$(C)
1070 WEND
1080 GOSUB 1180 ' check last keyword
1090 TW=TW+1     ' up word count for RND use
1100 RETURN
1110 '
1120 ' save words bracketed by the keywords
1130 '
1140 FOR I=0 TO KW
1150    IF KWD$(I)=W$ THEN 1180 ' it's a keyword
1160 NEXT
1170 PH$=PH$+" "+W$ : RETURN ' not a keyword, hold it in PH$
1180 IF PH$<>"" THEN TW=TW+1 : KWR$(TW)=MID$(PH$,2) : PH$="" ' save PH$
1190 RETURN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 3.2 ELSY - "Eliza-like" conversationalist

Word-Find Puzzles

There are innumerable types of *word puzzles* such as crosswords, acrostics, word-finds, jumbles, and so on. This section provides a word-find puzzle program generator. Puzzles will be created such as the one in Figure 3.3 where the following eight words, BASIC, PASCAL, COBOL, FORTRAN, FORTH, LISP, ALGOL, and MODULA (all computer languages), are hidden.

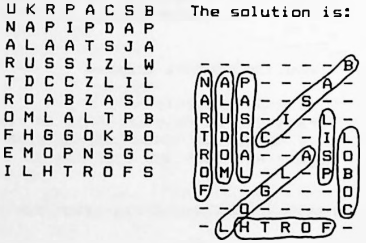
Our puzzle generator is used as follows. A DATA statement table contains a list of words that are to be used in the puzzle. The words are initially READ into the array WORDS$ for later use in the puzzle generation process. The variable NW is the count of the number of words read.

The program first asks you how many words you want to include in the puzzle. Enter a number from 1 to the total number of words in the table. This allows you to generate different puzzles with subsets of words that are included in the table.

The size of the puzzle is then requested. Enter the puzzle's height and width (number of rows and the characters per row) as two numbers separated by a comma. The size is currently limited to grids from 8- by 8- characters to 20- by 25 characters, but larger sizes can be entered by just changing the restriction in the program and changing the section that prints the puzzle on the printer.

The larger the size, the easier it is to fit all the words in the puzzle. For smaller puzzles, care should be exercised in making sure that the dimensions are at least big enough for the largest word to be included in the puzzle. Otherwise, large words in the table will be skipped over and excluded from the puzzle. The height and width values are stored in the variables H and W.

FIGURE 3.3. Sample word-find puzzle.



After the height and width are entered, the puzzle is generated. As the program works on the puzzle, it displays its work: When a word is added to the matrix, the puzzle's contents up to that point are printed on the screen. This gives you an idea of how the program is progressing in creating your puzzle. Figure 3.4 illustrates the start-up and conclusion for a 12- by 8-character puzzle.

FIGURE 3.4. Excerpt from the word puzzle program WPUZ.

```
WPUZ  -  Word Puzzle Generator


Loaded 23 words from within program.


Enter number of words you want in puzzle: 12

Enter height for puzzle (8-20)? 8

Enter width for puzzle (8-25)? 16


 1 Inserted COMPUTER after 1 try.

 - - - - C - - - - - - - - - - -
 - - - - - O - - - - - - - - - -
 - - - - - - M - - - - - - - - -
 - - - - - - - P - - - - - - - -
 - - - - - - - - U - - - - - - -
 - - - - - - - - - T - - - - - -
 - - - - - - - - - - E - - - - -
 - - - - - - - - - - - R - - - -


 2 Inserted MICRO after 2 tries.

 - - - - C - - - - - - - - - - -
 - - - - - O - - - - - - - - - -
 - - - - - - M - - - - - - - - -
 - - - - - - - P - - - - - - - -
 - - - - - - - - U - - - - - - -
 - - - - - - - - - T - - - - - -
 - - - - - - - - - - E O R C I M
 - - - - - - - - - - - R - - - -


 3 Inserted HARDWARE after 3 tries.

 - - - - C - - - - - - - - - - -
 - - - - - O - - - - - - - - - -
 - - - - - - M - - - - - - - - -
 - - - - - - - P - - - - - - - -
 - - - - - - - - U - - - - - - -
 - - - - - - - - - T - - - - - -
 - - E R A W D R A H E O R C I M
 - - - - - - - - - - - R - - - -
```

FIGURE 3.4. Continued.

```
12 Inserted ALGOL after 13 tries.

 -  -  -  -  C  -  -  -  -  P  -  -  -  -  -  -
 M  -  -  -  -  O  -  -  A  -  -  C  B  -  -  -
 A  -  -  -  -  -  M  S  B  Y  T  E  O  I  -  M
 R  B  A  S  I  C  C  P  L  O  G  L  A  B  T  O
 -  -  -  -  -  A  -  -  U  -  -  -  -  -  O  R
 -  -  -  -  L  -  -  -  -  T  -  -  -  -  -  L
 -  -  E  R  A  W  D  R  A  H  E  O  R  C  I  M
 -  -  -  S  O  F  T  W  A  R  E  R  -  -  -  -

A total of 12 words inserted in puzzle.

Print complete puzzle and solution (Y or N) ?

There are 12 words in this puzzle.

 Y K D Q C M P L K P H A T Z L B
 M H N J C O G U A L H C B I O H
 A P E D U V M S B Y T E O I N M
 R B A S I C C P L O G L A B T O
 E S G A U A N C U C Y F T Z O R
 W V V F L B U P G T G S T I S L
 L I E R A W D R A H E O R C I M
 Q A H S O F T W A R E R X R P O
```

When the puzzle is completed, the program asks if you want the puzzle and its solution outputted to the printer. Answer with a Y or N for yes or no. If answered no, the program begins again; if answered yes, it prints both the puzzle and solution on your printer.

Occasionally, the program will take quite a while to insert a word in your puzzle. If this happens, the program will stop and ask you the following:

```
The word PASCAL will not fit in the puzzle.
Abort, By-pass, or Continue trying (A, B, C) ?
```

Press either the A, B, or C key. Aborting the puzzle generation stops the puzzle and allows you to start the program over from the beginning. If you press B, then the word listed is skipped and the next word from the table is tried. Your third choice is to press C and let the program try to insert the current word in the puzzle again.

This message usually indicates that the matrix size is too small for the words being used (or the words are too large for the matrix). If you bypass many words and there are no more in the table, then the program may end with fewer words in the puzzle than you asked for.

When this happens, just rerun the program with a different-size matrix.

To create puzzles for other words, just replace the DATA statement table with your own set of words. Program 3.3 contains the listing of the complete word puzzle program WPUZ.

```
0 ' WPUZ   Word find puzzle generator
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  Use words from DATA statement table to create word find puzzles.
50 '
60 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
70 DEFINT A-Z : RANDOMIZE VAL(RIGHT$(TIME$,2)+MID$(TIME$,4,2))
80 PRINT"WPUZ  -  Word Puzzle Generator" : PRINT
90 RESTORE : READ NW : DIM WORDS$(NW)
100 FOR I=1 TO NW : READ WORDS$(I) : NEXT
110 PRINT : PRINT "Loaded" ; NW ; "words from within program." : PRINT
120 '
130 ' get and edit puzzle dimensions
140 '
150 PRINT : LINE INPUT "Enter number of words you want in puzzle: ",N$
160 WP=VAL(N$) : IF WP<1 OR WP>NW THEN PRINT"From 1 to";NW;"words" : GOTO 150
170 PRINT : LINE INPUT "Enter height for puzzle (8-20)? ",N$
180 H=VAL(N$) : IF H<8 OR H>25 THEN PRINT "From 8 to 20!" : GOTO 170
190 PRINT : LINE INPUT "Enter width for puzzle (8-25)? ",N$
200 W=VAL(N$) : IF W<8 OR W>25 THEN PRINT "From 8 to 25!" : GOTO 190
210 '
220 '  initialize puzzle character array
230 '
240 DIM P(H,W)     ' puzzle matrix
250 WT=0 : IL=99  ' total words inserted, insert try limit
260 '
270 '  build puzzle
280 '
290 FOR I=1 TO NW
300 W$=WORDS$(I) : L=LEN(W$) : IT=1 ' W$ is word being inserted
310 IF L>H AND L>W THEN 790 ' can word W$ fit in this matrix ?
320 '
330 ' pick starting postion and direction
340 '
350 PR=INT(RND*H)+1 : PC=INT(RND*W)+1  ' random starting row and column
360 DR=INT(RND*3)-1 : DC=INT(RND*3)-1  ' random row and column direction
370 IF DR=0 AND DC=0 THEN 360          ' dirction 0,0 not valid
380 IF PR+DR*L<0 OR PR+DR*L>H+1 THEN 350   ' exceeds row boundary
390 IF PC+DC*L<0 OR PC+DC*L>W+1 THEN 350   ' exceeds column boundary
400 '
410 '    can W$ be inserted at PR,PC in direction DR,DC ?
420 '
430    R=PR : C=PC
440    FOR J=1 TO L
450    A=ASC(MID$(W$,J,1)) : V=P(R,C)
460    IF V<>0 AND V<>C THEN IT=IT+1 : IF IT<IL THEN 350 ELSE 690
470    R=R+DR : C=C+DC
480    NEXT
```

```
490 '
500 '    word fits, so let's insert it
510 '
520    R=PR : C=PC
530    FOR J=1 TO L
540    A=ASC(MID$(W$,J,1)) : P(R,C)=A  ' insert character code
550    R=R+DR : C=C+DC
560    NEXT
570 '
580 '    display puzzle up to this point
590 '
600    WT=WT+1:PRINT:PRINT:PRINT WT;"Inserted ";W$;" after";IT;
610    IF IT=1 THEN PRINT "try." ELSE PRINT "tries."
620    FOR R=1 TO H : PRINT : FOR C=1 TO W : PRINT "  ";
630    IF P(R,C)<>0 THEN PRINT CHR$(P(R,C)); ELSE PRINT "-";
640    NEXT : NEXT : PRINT
650    IF WT<WP THEN 790 ELSE 830 ' more words to insert? or finished?
660 '
670 '  can't fit word W$; tell user
680 '
690     PRINT : PRINT "The word ";W$;" will not fit in the puzzle."
700     PRINT "Abort, By-pass or Continue trying (A,B,C)? ";
710     C$="" : WHILE C$="" : C$=INKEY$ : WEND
720     IF C$="A" OR C$="a" THEN 1030
730     IF C$="B" OR C$="b" THEN 790
740     IF C$="C" OR C$="c" THEN IT=1 : GOTO 350
750     BEEP : PRINT " Enter A, B or C" : GOTO 700
760 '
770 '    next word
780 '
790 NEXT I
800 '
810 '  Done, display puzzle and solution
820 '
830 PRINT : PRINT "A total of";WT;"words inserted in puzzle." : PRINT
840 PRINT "Print complete puzzle and solution (Y or N) ?";
850 C$="" : WHILE C$="" : C$=INKEY$ : WEND
860 IF C$="N" OR C$="n" THEN 1030
870 IF C$<>"Y" AND C$<>"y" THEN 850
880 PRINT"Turn on printer and press <SPACE> when ready..."
890 WHILE INKEY$<>" " : WEND
900 '
910 ' output puzzle to printer
920 '
930 LPRINT "There are";WT;"words in this puzzle." : LPRINT
940 FOR R=1 TO H : LPRINT : FOR C=1 TO W : LPRINT " ";
950 IF P(R,C)<>0 THEN LPRINT CHR$(P(R,C)); ELSE LPRINT CHR$(INT(RND*26)+65);
960 NEXT : NEXT
970 FOR I=1 TO 12 : LPRINT : NEXT ' space between puzzle and solution
980 LPRINT "The solution is:" : LPRINT
990 FOR R=1 TO H : LPRINT : FOR C=1 TO W : LPRINT " ";
1000 IF P(R,C)<>0 THEN LPRINT CHR$(P(R,C)); ELSE LPRINT "-";
1010 NEXT : NEXT
1020 LPRINT : LPRINT : PRINT "All done."
1030 PRINT : PRINT "Would you like to create another puzzle (Y or N)? ";
1040 C$="" : WHILE C$="" : C$=INKEY$ : WEND
1050 IF C$="Y" OR C$="y" THEN CLS : ERASE P : GOTO 110
1060 IF C$="N" OR C$="n" THEN C$=CHR$(27) : GOTO 9050
```

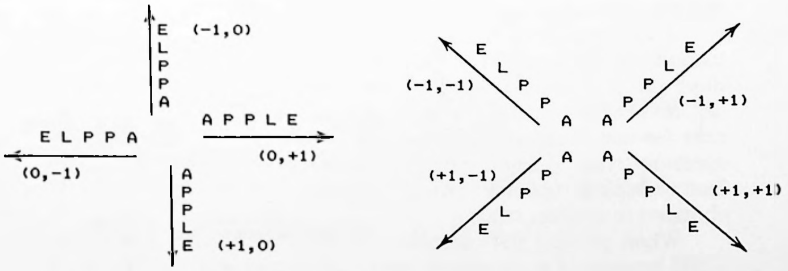
```
1070 BEEP : GOTO 1040
1080 '
1090 ' words to insert in puzzle
1100 '
1110 DATA 23
1120 DATA COMPUTER, MICRO, HARDWARE, SOFTWARE, BIT, BYTE, RAM, ROM
1130 DATA BASIC, PASCAL, COBOL, FORTRAN, ALGOL, FORTH, LISP, MODULA
1140 DATA ASCII, DATA, FILE, RECORD, MODEM, DISPLAY, PRINTER
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 3.3 WPUZ - Word puzzle generator

The puzzles are created as follows. The program's overall approach is to take each word from the table in succession, randomly select a potential place for each word in the matrix, and then insert the word there, if possible. A word may be inserted either vertically or horizontally, with its letters forward or backward. A word may also be inserted diagonally in either direction. Figure 3.5 illustrates the eight possible directions.

The pair of numbers associated with each direction refers to the manner in which the row and column indices of the puzzle matrix are incremented. Thus, a word that is displayed left to right increments with the number pair 0, + 1: The row is not changed and the column

FIGURE 3.5. Directions for the insertion of words in word puzzles.



increases by one. A value of 0,0 is not valid (there is no change in either direction).

The two-dimensional array P contains the integer character codes (65 to 90 for the letters A to Z) of the words in the puzzle. Use of numeric character codes causes the program to run faster, than if individual character strings were used. Recall that BASIC's ASC and CHR$ allows us to convert back and forth from the ASCII character codes to their character equivalents.

The random starting point in the array is calculated by multiplying the random number function, RND, by the height and width variables H and W for the puzzle. The resulting point is stored in the variables PR and PC for the row and column, respectively. A random direction, as illustrated earlier, is calculated in a similar manner and stored in variables DR for the row direction, and DC for the column.

These variables are used to step through the matrix while examining its contents. The variables R and C are set to PR and PC. They are then incremented by DR and DC, which causes each to change in the appropriate random direction.

Based on the length of the word being inserted, the computer extracts integers along the path from point PR,PC in the direction DR and DC. These integers are compared with the corresponding ASCII codes for the characters in the variable W$, the word being inserted.

An integer value of 0 in the matrix means that any letter is allowed at that point; otherwise, the letter at that position would have to match the corresponding letter in the word being inserted. Any nonzero mismatch excludes that word from being used, and a new matrix position and direction are chosen. If a word fits, it is inserted and the process begins again with the next word.

If the word does not fit into the allotted space, the program increments the counter, IT, and tries another matrix position and direction. If IT exceeds IL, the "insert try limit," then the program assumes that things are too crowded in the matrix for this word and asks the user if he wants to abort the puzzle, bypass this word, or continue trying to insert the word as described earlier. The insert limit variable is currently set at 99 in the program, but it can be easily changed to another value.

When printing the completed puzzle, the program prints the CHR$ function of each nonzero entry in the puzzle array. For the 0 values, a random letter is chosen and printed. The solution lists just

the nonzero values (since they are the letters in the words that are hidden) and prints a dash for the zero elements.

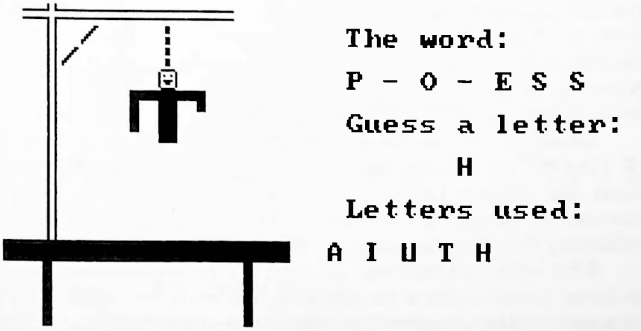
There are numerous alternatives to the above approach of inserting a word. For instance, after picking the random starting point and direction for a potential word in the matrix, all words not currently used could be tested to see if they fit in the space. You would pick a position and then find a word for it instead of picking a word and finding a position for it. This is essentially the opposite of the above approach. Its advantages or disadvantages are left for you to determine.

A Hangman Game

Although not quite a classic, the game of Hangman is frequently found on computer systems. Our version of the game will include a character graphics version of the gallows and the "man" being hung as the guesses are made. See Figure 3.6 for an illustration of the game being played.

To play the game, make your guesses by pressing the chosen letter key (it's not necessary to also press the Enter key). The program converts each letter to upper case for you and ignores all nonletters except the question mark and the Esc key. Press the ? key to make a full-word guess; for example, if you think you know the word and want to key it in. Press the Esc key if you want to exit from and end the game early.

FIGURE 3.6. Hangman game example.



As you make your single-letter guesses, each letter that is found in the word is inserted. If a letter is not found in the word, that letter is displayed in a list below where you key in the guesses, and a part of the man's body is displayed. You are allowed up to ten incorrect letter guesses, after which the man is hanged and the game is over. You are allowed only one complete word guess (after pressing the ?).

A DATA statement table in the program contains a set of words that are used by the game. One word from the DATA statement table is randomly selected each time you play. To put your own words in the program, just delete the data that is currently there or add to it with your own list of words. A count of how many words are in the list must be the first item in the table.

The program for the hangman game works as follows. The program first selects a word to use by reading a random number of words from the table—the last word that is read is the word used in the game. Next, the gallows and the head of the man are displayed. A row of dashes is printed, alternating with spaces, for each letter in the word to be guessed.

When accepting your guesses, the program makes use of the INKEY$ function. As your keystrokes are made, the program first checks for the Esc character or the question mark. If a lower-case letter is entered, it is converted to upper case. The program then validates the character by checking if it is an alphabetic letter and if it has not been used in an earlier guess. If either of these checks fail, then the program sounds a tone and asks again for your guess.

A FOR / NEXT loop is used to examine each letter in the word. If a match is made, the program sets a "hit" variable, HIT, and displays the letter on the screen in that position for the word. All letters in the word must be checked because there may be duplicates such as double o's or e's. If there is no hit, then the letter is displayed on the screen in an area of previously used letters, the guess count is upped by one, and another portion of the man's body is displayed.

All letters that have been used are saved in the string variable L$. Note that L$ includes letters that are found in the word as well as those that are not. Letters guessed but not found in the word are counted by variable LU. If this count exceeds 10,the game is over—indicating that the player has not been able to guess the word.

If the letter that has been guessed is inserted into the word, the program checks if this is the last letter needed to complete the word by counting the number of hits (letter guesses that were correct) with

the variable LC. If LC equals the length of the word to be guessed, then every letter has been found and the game is over. If any letters have not been found, the program continues and the player makes another guess.

When the player makes a full-word guess (after entering a ? instead of a single letter), the program performs a LINE INPUT. The player may now enter his guess for the complete word. The program converts each letter to upper case and makes a comparision with the secret word. If the two words match, the player wins; if the words do not match, the player loses and the game is over.

As the full-word guess is entered, the program has no special control over the text entered by the player. This means that use of the cursor control keys, for example, is not disabled and the player can move the cursor around the screen if he wishes. To circumvent this requires a character-by-character read from the keyboard using the INKEY$ variable. (This is left up to you to develop if you feel it is necessary.) The complete listing for the program is Program 3.4, HMAN.

```
0 ' HMAN   Hangman game
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  Play the word guessing game hangman.
50 '
60 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
70 DEFINT A-Z : RANDOMIZE VAL(RIGHT$(TIME$,2))
80 PRINT "HANGMAN Word Guessing Game" : PRINT
90 RESTORE : READ NW ' get number of words
100 PRINT "You have 10 guesses, then its all over!"
110 PRINT "Press ? to enter whole word as a guess."
120 PRINT : PRINT "Press <SPACE> to begin game";
130 WHILE INKEY$<>" " : WEND ' wait for space
140 CLS
150 '
160 ' display the gallows
170 '
180 PRINT SPC(4);CHR$(205);CHR$(206);STRING$(9,205);       ' top
190 FOR R=2 TO 11 : LOCATE R,6 : PRINT CHR$(186); : NEXT   ' column
200 LOCATE 12,4 : PRINT STRING$(15,219);                   ' base
210 FOR R=13 TO 15
220 LOCATE R,6 : PRINT CHR$(221);SPC(9);CHR$(222)          ' legs on base
230 NEXT
240 LOCATE 2,7 : PRINT" /    |";  ' the rope
250 LOCATE 3,7 : PRINT"/     |";
260 LOCATE 4,12: PRINT CHR$(1);  ' smiley face for head
270 '
280 ' pick a word and print screen prompts
290 '
300 RESTORE 970 : FOR I=1 TO INT(RND*NW)+1 : READ W$ : NEXT  ' random word
```

```
310 LOCATE 2,22 : PRINT "The word:";
320 LOCATE 4,22 : FOR I=1 TO LEN(W$):PRINT"- ";:NEXT  ' dashes for each letter
330 LOCATE 6,22 : PRINT "Guess a letter:";
340 LOCATE 10,22: PRINT "Letters used:";
350 '
360 ' play the game
370 '
380 L$="" : LC=0 : LU=-1 ' letter count and how many used so far
390 LOCATE 8,28,1
400 C$=INKEY$ : IF C$="" THEN 400  ' get player's letter guess
410 IF C$=CHR$(27) THEN 9030       ' press Esc key to exit from game
420 IF C$="?" THEN 730             ' guess full word on question mark
430 C=ASC(C$)
440 IF C>96 AND C<123 THEN C=C-32 : C$=CHR$(C) ' change to upper case
450 IF C<65 OR C>90 OR INSTR(L$,C$) THEN BEEP : GOTO 400 ' validate letter
460 '
470 ' check if letter is in word
480 '
490 HIT=0 : PRINT C$; ' echo player's guess
500 FOR I=1 TO LEN(W$)
510 IF MID$(W$,I,1)=C$ THEN LOCATE 4,(I-1)*2+22 : PRINT C$; : HIT=-1 : LC=LC+1
520 NEXT
530 L$=L$+C$ ' save letter as in use
540 IF LEN(W$)=LC THEN 650 ELSE IF HIT THEN 390 ' word guessed?
550 '
560 ' letter not in word so display as used, and hang more of man
570 '
580 LU=LU+1 : LOCATE 12,21+LU*2 : PRINT C$; ' letter in use
590 ON LU+1 GOSUB 830,840,850,860,870,880,890,900,910,920 'display parts of man
600 IF LU<9 THEN 390  ' any more guesses ?
610 '
620 ' end of game
630 '
640 LOCATE 18,1 : PRINT"Sorry! The word was: ";W$ : GOTO 660
650 LOCATE 18,1 : PRINT"Good going!  You got it!!"
660 PRINT: PRINT "Press Esc to exit or SPACE to play again";
670 C$="" : WHILE C$="" : C$=INKEY$ : WEND
680 IF C$=" " THEN CLS : GOTO 180 ELSE IF ASC(C$)=27 THEN 9050
690 BEEP : GOTO 670
700 '
710 ' process whole word guess (convert to upper case before comparison)
720 '
730 LOCATE 14,21 : PRINT "Guess the word:"
740 LOCATE 16,22 : LINE INPUT "",G$
750 IF G$="" THEN BEEP : GOTO 740
760 FOR I=1 TO LEN(G$)
770 C=ASC(MID$(G$,I,1)) : IF C>96 AND C<123 THEN MID$(G$,I,1)=CHR$(C-32)
780 NEXT
790 IF G$<>W$ THEN 640 ELSE 650 ' game ends after guess
800 '
810 '  "hang" successive portions of man
820 '
830 LOCATE 5,12 : PRINT CHR$(219);:RETURN
840 LOCATE 5,10 : PRINT CHR$(222);CHR$(223);:RETURN
850 LOCATE 5,13 : PRINT CHR$(223);CHR$(221);:RETURN
860 LOCATE 6,12 : PRINT CHR$(219);CHR$(31);CHR$(29);CHR$(223);:RETURN
870 LOCATE 6,10 : PRINT CHR$(222);:RETURN
880 LOCATE 6,14 : PRINT CHR$(221);:RETURN
```

```
890 LOCATE 7,11 : PRINT CHR$(222);:RETURN
900 LOCATE 7,13 : PRINT CHR$(221);:RETURN
910 LOCATE 8,11 : PRINT CHR$(222);CHR$(31);CHR$(29);CHR$(223);:RETURN
920 LOCATE 8,13 : PRINT CHR$(221);CHR$(31);CHR$(29);CHR$(223);:RETURN
930 '
940 '  word table; first entry is how many
950 '
960 DATA 24
970 DATA SYSTEM    , DATA      , INFORMATION
980 DATA COMPUTER , HARDWARE , SOFTWARE
990 DATA INPUT     , OUTPUT    , PROGRAM
1000 DATA MEMORY    , BYTE      , BITS
1010 DATA DATABASE , MICRO     , PROCESSOR
1020 DATA BASIC     , PASCAL    , COBOL
1030 DATA LOGIC     , NUMERIC   , ALPHABETIC
1040 DATA COMPILER , LANGUAGE , INTERPRETER
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 3.4 HMAN - Hangman game

Variations

Some ideas for changing the word games are listed below.

1. Expand the random-sentence generator to include other subjects. Add the words "or" and "and" to create compound sentences.

2. Add more random-reply sentences to Elsy, incorporating some first-person names of potential users of the program (children, friends, other family members, and so on).

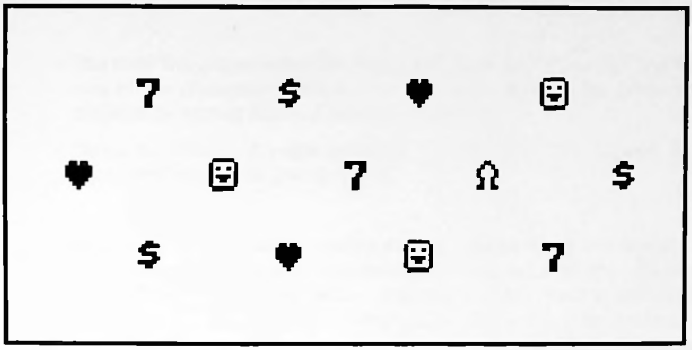
3. Add a second insert spot in some of Elsy's replies. The program must then check that at least two keyword phrases are found in the user's sentence and if the random reply has two blanks in which to insert them.

4. The word-puzzle generator can be modified for different word-insertion techniques, as previously mentioned. Other approaches might include removing previously inserted words to make room for others. After a certain number of attempts at inserting a word, the program could go into a point-by-point examination of the puzzle matrix instead of randomly choosing starting positions.

5. The matrix can be modified to create puzzles other than in a rectangular shape. Just add another character code in those positions where no letter is to be inserted. Then have the program ignore these positions when attempting to insert a word. For example, you might want to create a puzzle in the shape of a triangle or a diamond.

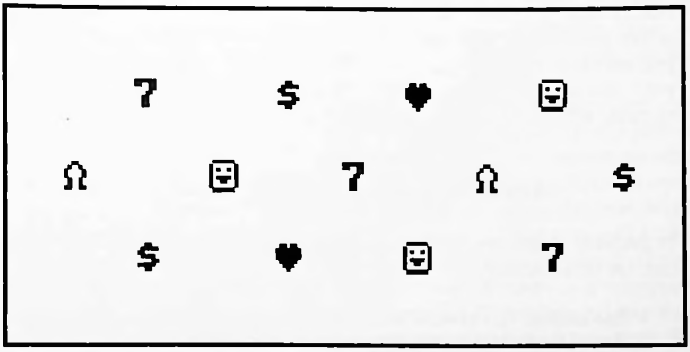
6. In the Hangman game, many variations are possible: Use phrases instead of guessing individual words; provide random blanks in the words so the number of dashes is not necessarily the length of the word being guessed; do not show any dashes so the length of the word is hidden from the player.

7. Add color to the displays in the Hangman game: the gallows might be one color; parts of the man other colors; and the text a high-intensity white. Add music to the game: If a guess is correct, play a short snappy song; if a guess is incorrect, play a somber tune.



4

An Adventure Game



With adventure games, you enter a make-believe world where your imagination is the only limit to what can happen. Often called *role-playing* games, adventure games allow players to take on a new personality and become a new being as they explore a castle, dungeon, medieval land, or village. Sorcery, magic, strange beings, and assorted weird happenings are the norm.

A number of such games are played on a board that is often made in some ad hoc manner by a game master (director and referee). Players possess different characteristics such as strength, fortitude, charisma, and intelligence which, along with the roll of the dice, determine how they fare in the game—or how they *campaign*. As points are earned, your character and the game evolve over time into a completely new world.

Adventure games played on a computer are somewhat like a word maze, a puzzle, or an interactive novel in which you play an active part in the outcome of the story. In your make-believe world, you set off to explore, and discover an object or a person in need of your rescue. You interact with the game by entering word commands, which the game analyzes and acts upon accordingly.

ADVENTURE-GAME CHARACTERISTICS

In general, there are three characteristics of an adventure game that can be considered:

- The setting: This is essentially the world that the player enters. It consists of the background scenery, other characters, and objects that the player may interact with in various ways.

- The task: The player enters the world to explore, find some object, save one of the characters, and so on. This is the goal of the game (in addition to having fun and being entertained).
- Game interaction: A major part of the game is how you achieve the goal—half the fun is getting there!

The setting for the game is usually preset before the game begins. Each time you play, you will find yourself in the same setting. Placement of the objects and the other characters in the new world may also be fixed. The more you play the game, the more you will learn about the proper path to take in exploring your world and how to handle each of the characters you meet. Even when the goal of the game is achieved (which can take many hours), you may play the game again to explore other areas or to improve the time it takes to achieve the goal of the adventure.

Alternatively, the setting can be preset but with random placement of some or all of the objects and characters within it. Thus, as you play, you will learn the terrain and what to do against each adversary, but not necessarily when and where to expect them. A dragon might be hiding in a cave during one play or out in the forest the next time the game is played. This leads to more open-ended play where each game is practically a new campaign.

It is also possible to create a new random setting each time the game is played. In this case, each game is completely new where the setting, characters, and objects are different: A list of the places, objects, and characters are maintained, and a set of rules are used for creating a random setting each time. Even the information of what kind of action must be taken when a character is met can be based on a random value.

The manner of interacting with the setting or environment can also be presented in a number of different ways. Usually, the player has a limited vocabulary of words that are used in conversing with the program. There are words or commands for moving from place to place, for using the objects found, and for handling the inevitable confrontations that occur between the player and an an evil force.

Most adventure games allow entry of one or two words at a time; other games allow the use of more involved phrases and complete sentences. Learning the vocabulary may be from trial and error as the game unfolds, or the allowable words might be given from the beginning so the player can quickly get into the action. A game

might also learn from its player new words and how to use them, such as using synonyms in lieu of a particular phrase.

In addition to moving about in the setting, there are usually words for taking and using the objects found, for going into battle with any adversaries, and for viewing the environment. The latter are usually given as textual descriptions in reply to the player's commands. A number of adventure games provide graphic images of game play; as the setting changes, a new image is presented. The player actually sees the characters and objects that confront him.

So much for generalities. It's time to develop an adventure game of your own. Our game is called Igor Brimski manor, a haunted house that you can explore.

Igor Brimski Manor Adventure Game

Remember the last time you were in a haunted house? It was a horrible experience, no doubt. Well, for our adventure game program, we have the deserted and mysterious haunted house of Igor Brimski for you to explore. It's a pretty bad place—full of witches, ghouls, goblins, and assorted other creatures of the night. Your job is to explore the manor house, evading the creatures, and finding treasures (gold and silver), and things that will help in your exploration (a magic wand or some weapon, for example). If you take too much treasure, you will not be able to take any weapons to defend yourself—you are limited to a maximum of five objects in your possession at one time.

As an overall goal, you must find a fair damsel who is hidden in the house. The game scores how well you do based on the value of the treasures you take and whether or not you have found the fair damsel. If you encounter some evil creature, such as a demon or warlock, you may have to defend yourself. If your defense is not good enough, you will lose and the game will be over; if you defend yourself successfully, you continue to explore the house by moving from room to room and floor to floor.

The game is designed to allow a virtually unlimited number of settings. Each time you begin play, you first enter a "house number," which is a value from 1 to 999. The program uses this number to create the setting: the number of floors in the house, the number of rooms, which objects are in which rooms, and the points associated

with each object. Thus, if you play with house number 707, then stop playing and return to play later, you can use 707 again and have the same setting to deal with. Otherwise, you can enter another number and a whole new "haunted house" will be created for you.

Each house is structured as follows. It may have from one to five floors; each floor may have from two to six rooms. A hallway divides each floor in half with a stairway in the middle for moving up and down floors. There are from one to three rooms on each side of the hallway. There may be from one to five "objects" within a room.

Objects fall into three classes: There are evil creatures which might attack you, such as a ghoul or a werewolf; there are treasures, such as gold and silver; and there are weapons that you can use to protect yourself. Note that you can only score points for taking treasures. In one of the rooms is the fair damsel whom you must find. Upon finding her, you are awarded a bonus of one hundred points. You end the game by moving to the first floor of the house and leaving through the front door.

To play the game, first enter a house number between 1 and 999. After this, the program pauses to create the house; then you begin actual play by entering your commands. The program responds only to those things it understands; otherwise, it says nothing. There are fourteen commands used in the game. Up to fifteen commands and words may be entered by the user at one time: They are each processed and acted upon in sequence. The following commands are recognized:

1. MOVE—Advance or MOVE in the direction that you are currently pointing.
2. RIGHT, LEFT, BACK—Change direction either to the RIGHT or LEFT, or turn around BACKward.
3. UP, DOWN—Go UP or DOWN the stairs to the next floor in the house.
4. ENTER, LEAVE, QUIT—Go into or ENTER a room, exit or LEAVE a room, and stop playing or QUIT the game.
5. LOOK—Take a look around and see what lies ahead of you.
6. TAKE, USE—These commands require an object (second word) which you TAKE from a room to keep or USE when fighting an adversary.
7. INVEN, VOCAB—Two commands to let you know what items you currently have in your personal INVENtory and what the overall VOCABulary of words is for the game (for example, this list).

Thus, a command to move down the hallway is simply:

```
MOVE
```

A command to take a wand from a room is:

```
TAKE WAND
```

To enter a room and see what's there, key in the two commands:

```
ENTER LOOK
```

See Figure 4.1 for an example of a session with the game. In this example, the player asked for manor house number 213, moves up to the second floor, and sees two doorways. Upon entering one of the rooms, a witch attacks the player.

The game uses the RND function to create a new random setting based on the house number. To enable a house to be recreated, the house number is used in a RANDOMIZE statement so that the same sequence of random numbers is generated each time when the house is set up. Thus, with the house number in variable H, the statement

```
RANDOMIZE H
```

is executed prior to performing a GOSUB to the routine that builds this particular manor house.

Once the house number and setting are created, the game begins with the program accepting the player's commands. The commands are separated into individual words with a GOSUB made to a subroutine for processing each word. If a word is not recognized, it is ignored and the next word is processed until there are no words left. The next set of commands from the player is then read. The program's response to your commands depends on the house that has been set up. There are many different combinations of floors, rooms per floor, objects per room, and so on.

To understand further how the program works requires an understanding of how the house is created. The initial phase of the setup consists of loading three arrays:

FIGURE 4.1. Example of a session with the Manor adventure game.

```
MANOR - Adventure Game

You are standing before the mysterious and dreaded Igor Brimski
manor.  Look above the doorway.  What house number do you see? 213

Excuse me for a moment while I set up the house...

You hear a loud scream!  A fair damsel in distress inside
the deserted manor house number 213 of Igor Brimski.

The house has 135 points of evil, but 45 treasure points,
and 125 weapon points hidden inside.

There isn't much time to save her from the evil that lies within.
You enter the doorway, and stand in a hallway before a staircase...


look

You are in a hallway containing 3 doorways

up look

You went up the stairs to floor 2
You are in a hallway containing 3 doorways

move

You're at the end of the hallway

left move

You have just turned to your left
You are standing before a doorway.  Dare you enter?


enter

You are standing in the middle of a room.
A cold chill runs down your spine...
There is evil lurking about...
There is much evil in the room !!!
Watch out! A witch shrieks at you!!!
The witch fumes at you!

look

You look about and see a witch, a demon, a gold, a ghost
The witch spits blood at you!

take gold

You now have 25 points
The witch fumes at you!
```

- KW$(i)—Keywords, or commands, used to play the game. The order of the keywords defines the GOSUB used to process the words. There are fourteen keywords loaded into this array.
- OBJ$(i)—Names of objects that may be placed in the house.
- TYP(i)—Type of object placed in the house: 1 is for weapons, 2 for treasures, and 3 for evil forces (adversaries). This array also controls how each object may be used by a player. For instance, you can TAKE a WAND, but you cannot TAKE a WITCH.

Note that the DATA statement table for the objects and their type are the last few lines in the program. This is done so that the program can load the objects and types into the array OJB$ and TYP without the number of them being known ahead of time. The program continues to read the object name and its type code until an "Out of DATA" error occurs, which is error number 4. An ON ERROR GOTO statement has been set prior to the reading which causes a GOTO to the part of the program that follows the error. The program assumes that there are no more than one hundred objects. Thus, it is easy to add more objects of your own to the program without keeping track of how many have been added or how many objects are already there.

The number of floors in the house is set randomly at a value between two and five. This is stored in variable NF. Then, for each floor, the number of rooms is randomly set to between two and six, which is stored in the array MF(i) where the variable i is an index to the floor number. Two other arrays are now created, which contain the objects in each room and the number of "points" associated with them:

- MM(i,j,k)—Contains an index, or pointer, to the OJB$ array for the kth object in the jth room on the ith floor. Although a three-dimensional array might appear somewhat cumbersome, it holds the information needed very neatly. For example, if

```
MM(1,2,3)=4
```

- it means that on floor 1 in room 2, the third item there has an object number of 4. Look up the number 4 in the OBJ$ and TYP arrays for the object's name and type. The number of objects in each room is at the zero position in the MM array, or MM(1,2,0) in the above example.

- MR(i,j,k)—Structured identically to MM but contains the point values for the objects in the rooms. Thus,

  MR(1,2,3)=25

  would indicate the object number 4, described earlier, has a point total of 25.

Points are used to indicate the following: A negative point value indicates an evil force, such as a ghoul or demon; a positive value greater than 100 is a treasure. The player may take treasures to score points. The "point values" between 1 and 99 are a measure of how powerful a particular weapon is. It determines how effective the weapon is when used in battle against an evil force.

Once a manor house is set up, it is a simple matter to list its contents. Of course, doing so is like telling you the answer to a riddle you must guess, but it's interesting to play in a house and then see what was in some of the rooms you did not enter. After the program starts, it can be interrupted by pressing the control break key (Ctrl Break). A GOTO 2060 will then cause transfer to a portion of the program where a description of the house is printed by listing the contents of the arrays MF, MM, and MR. See Figure 4.2 for an example of the contents of house 213.

The player may have up to five objects that he or she can TAKE from a room. Two arrays are used to keep track of the player's objects:

FIGURE 4.2. Descriptive listing of house number 213.

```
House number 213 has 2 floors.

  Floor 1 has 3 rooms.
  Room 1 has 2 objects, with points of 5 evil, 0 treasure, and  10 power
         mummy  -5, wand  10,
  Room 2 has 2 objects, with points of 10 evil, 0 treasure, and  10 power
         dagger  10, mummy -10,
  Room 3 has 2 objects, with points of 25 evil, 40 treasure, and  0 power
         silver  40, gargoyle -25,

  Floor 2 has 3 rooms.
  Room 1 has 4 objects, with points of 75 evil, 25 treasure, and  0 power
         witch -20, demon -30, gold  25, ghost -25,
  Room 2 has 4 objects, with points of 20 evil, 60 treasure, and  5 power
         warlock -20, gold  40, portrait  20, crossbow  5,
  Room 3 has 3 objects, with points of 0 evil, 0 treasure, and  20 power
         whip  10, chain  10, fair damsel  0,
```

- PITM$(i)—Description of an item, a weapon or treasure. It is a copy of the text from OBJ$ for the object taken from the room.
- PIM(i)—Points associated with a weapon. If an object is a treasure, its points are saved in the variable PTS and the value in PIM is set to 0. This allows the program to prevent the player from using a treasure to fight an adversary or evil object in a room.

Four other arrays are used in the program for miscellaneous messages displayed during the course of play. They are:

- ATKS$(i)—List of adversary attack messages.
- CRYS$(i)—Set of messages used when an adversary "cries out" at the player (during a fight, for example).
- WINS$(i)—Messages that describe how a player won a fight.
- LOSS$(i)—Messages that describe how a player lost a fight.

Last, there is the array PW$(i), which contains the words extracted from a player's command-line entry. The subroutine from the ELSY program (which did the same type of processing) is used. Once the list of words is extracted, each word is compared to the recognized keyword commands in the KW$ array. The position of the words in this array is used as a pointer to the list of subroutines that process each word via an ON position index GOSUB 520, 670, . . . for each of the fourteen keywords. (See line 360 in the program for a complete list of the subroutines used.)

After a keyword is processed, the variable PX is set to indicate whether to continue with the next word in the player's commands or to ask for new commands. The variable PX asks for new commands whenever the player bumps into a wall, encounters evil, or does something the program does not understand.

As players move about the house, the following variables describe their overall status:

- PD—The player's direction, where 0 is north, 1 is right, 2 is south, and 3 is left.
- PF—The floor number that the player is currently on.
- ROOM—The current room number (from 1 to 6), once again, depending on how many rooms are on a floor. ROOM may also be negative where −1 is the front end of the hall, −2 the middle, and −3 the back end of the hall.

- PTS and PWS—The current total of points and power the player has.
- PI—The total number of items the player has in his inventory. Note that when an object is used, it is removed from the inventory.
- Since a player may use an object picked at random, a 0 and a null entry in the array's PIM and PTIM$, respectively, indicate a vacant spot to which another object can be added. Thus, PI is not an index to the next available slot in these arrays for new objects.
- PW and NPW—These are pointers to the words in the array PW$ and the number of words in the array.

Processing of each keyword is fairly straight-forward. The words UP, DOWN, LEFT, RIGHT, and BACK merely change the variable PF (the floor number) or PD (the player's direction). The words INVEN and VOCAB list values in the arrays KW$ and PITMS$, respectively. The word LEAVE clears the variables for the room that was previously entered. The word QUIT simply transfers control to the exit routine for the game. The words LOOK, MOVE, TAKE, USE and ENTER are a little more complex.

Use of the word LOOK depends on whether a player is in the hallway or a room. If the player is in a hallway, he can see the stairs and how many doors there are. When the player is in a room, LOOK lists the objects that are there based on the values in the MM array for that particular floor and room.

The MOVE command is used for moving a player through the hallway. The player's direction is north and south from one end of the house to the other and left or right to face a doorway before entering a room.

The TAKE command must be followed by a word that identifies an object that can be seen. If an object is found in the room, it is added to your inventory in the arrays PITM$ and PIM (if you have not already taken five objects). The program also tells you what an object is worth if it is a treasure, or its power if it is a weapon. Note that once you take a treasure, you cannot give it up; this limits the number of weapons that you can take in order to defend yourself.

The USE command must also be followed by a word that identifies a weapon listed in your inventory of items (located in arrays PITM$ and PIM). The USE command is primarily used when the player is in battle with an adversary. The outcome of the battle depends on the power of the weapon and the strength of the adversary (based on its number of "evil" points). If the player is not doing battle

and uses a weapon, this is considered a "wasted" action since the player was not being attacked. This may also be used by the player to discard a weapon.

The ENTER command moves the player from the hallway into a room. Once the player is in a room, three things can happen: If there is no evil, an innocuous message is printed; if an evil creature with a value of less than twenty points is present, then a more foreboding message, such as "There is evil lurking about," is displayed; and if the evil creature has a value of more than twenty points or if a random value check is true, then one of the evil creatures attacks the player with a message from the ATKS$ array.

When the player is attacked, the variables AD and AE are set to indicate attack and the number of points of the attacker, respectively. When the player with the USE command fights back, the value of the object used is compared to AE to decide who wins the battle. The variable AD is also used by the program to display messages between the processing of the player's keywords. This is done to give the player a sense of urgency in fighting back the adversary.

The complete listing for this game is shown in Program 4.1, MANOR.

```
0 ' MANOR   Igor Brimski Manor "adventure" game
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  Adventure game: Find and save the fair maiden who is in the
50 '  mysterious and horrible Igor Brimski manor.
60 '
70 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS : DEFINT A-Z
80 DIM KW$(15), PW$(15), OBJ$(99), TYP(99)
90 DIM MF(5), MR(5,6,6), MM(5,6,9), PIM(5), PITM$(5)
100 DIM CRYS$(9), ATKS$(9), WINS$(9), LOSS$(9)
110 PRINT "MANOR - Adventure Game" : PRINT
120 PRINT "You are standing before the mysterious and dreaded Igor Brimski"
130 PRINT "manor.  Look above the doorway.  What house number do you see? ";
140 LINE INPUT H$: H=VAL(H$): IF H>0 AND H<1000 THEN 170
150 PRINT: PRINT "Whoops!  Manor houses are numbered from 1 to 999.  Check"
160 PRINT "that number again and enter what you see: ";: GOTO 140
170 PRINT : PRINT "Excuse me for a moment while I set up the house..." : PRINT
180 RANDOMIZE H : RESTORE : GOSUB 2240
190 PRINT "You hear a loud scream!  A fair damsel in distress inside"
200 PRINT "the deserted manor house number";H;"of Igor Brimski." : PRINT
210 PRINT "The house has";GTE;"points of evil, but";GTM;"treasure points,"
220 PRINT "and";GTP;"weapon points hidden inside." : PRINT
230 PRINT "There isn't much time to save her from the evil that lies within."
240 PRINT "You enter the doorway, and stand in a hallway before a staircase..."
250 PRINT
260 PI=0 : PF=1 : PTS=0 : PWS=0 : FEM=0 : GOSUB 1910 ' initial player variables
270 '
```

```
280 ' get player's entry, parse it, and act accordingly
290 '
300 PRINT
310 LINE INPUT PE$ : GOSUB 1960 ' separate into single words
320 PRINT
330 FOR PW=1 TO NPW
340   FOR KW=1 TO NKW
350   PX=0 ' early exit flag
360   IF PW$(PW)=KW$(KW) THEN ON KW GOSUB 520,670,720,770,800,830,860,1060,1130,
 1250,1310,1500,1570,9030
370   IF PX=1 THEN 400 ELSE IF PX=2 THEN 450
380   NEXT
390 NEXT
400 IF AD>0 THEN PRINT "The ";OBJ$(AD);" ";CRYS$(INT(CRY*RND));" at you!"
410 GOTO 300
420 '
430 ' end of game
440 '
450 PRINT : PRINT "Your adventure is over!  You have ";
460 IF FEM THEN PRINT "saved the fair damsel!  How brave!!!"
470 IF NOT FEM THEN PRINT "left the fair damsel in the manor house!"
480 PRINT "You scored";PTS;"points on this adventure":PRINT:GOTO 9030
490 '
500 ' process a keyword
510 '
520 '****** LOOK ********
530 IF ROOM>0 THEN 550
540 PRINT "You are in a hallway containing";RMS;"doorways" : RETURN
550 N=0 : DA=0 : PRINT "You look about and see ";
560 FOR I=1 TO MR(PF,ROOM,0)
570 IT=MR(PF,ROOM,I) : IF IT=99 THEN DA=I
580 IF IT=0 THEN 600 ELSE IF N<>0 THEN PRINT ", ";
590 N=N+1 : ITM$=OBJ$(IT) : GOSUB 1860
600 NEXT
610 IF N=0 THEN PRINT "nothing.  The room is empty!" ELSE PRINT
620 IF DA=0 THEN RETURN
630 PRINT "Good going!  You have found the fair damsel."
640 PRINT "Quickly now... you must leave the manor house!"
650 FEM=-1 : PTS=PTS+100 : MR(PF,ROOM,DA)=0 ' set flag and remove from house
660 RETURN
670 '****** UP ******
680 IF ROOM<>-2 THEN PRINT "You aren't next to the stairway":PX=1:RETURN
690 IF PF=NF THEN PRINT "Can't go on the roof!":PX=1:RETURN
700 PF=PF+1 : PRINT "You went up the stairs to floor";PF
710 GOSUB 1910 : RETURN
720 '****** DOWN *****
730 IF ROOM<>-2 THEN PRINT "Need to move to the stairs first":PX=1:RETURN
740 IF PF=1 THEN PRINT "The manor house doesn't have a basement":PX=1:RETURN
750 PF=PF-1 : PRINT "You went down the stairs to floor";PF
760 GOSUB 1910 : RETURN
770 '****** LEFT *****
780 PD=PD-1 : IF PD<0 THEN PD=3
790 PRINT "You have just turned to your left":RETURN
800 '****** RIGHT ****
810 PD=(PD+1) MOD 4
820 PRINT "You have turned to the right":RETURN
830 '****** BACK *****
840 PD=(PD+2) MOD 4
```

```
850 PRINT "You just turned around":RETURN
860 '***** ENTER *****
870 IF DOOR>0 AND ROOM<0 THEN 910
880 IF ROOM>0 THEN PRINT "Enter what?  You're already in a room":PX=1:RETURN
890 PRINT "Thump!  You just bumped into a wall.  How about"
900 PRINT "moving in front of a door before you enter.":PX=1:RETURN
910 PRINT "You are standing in the middle of a room."
920 ROOM=DOOR : EV=MM(PF,ROOM,7)
930 IF RND<.3 THEN PRINT "A cold chill runs down your spine..."
940 IF RND(0)>.3 AND RND(0)<.6 THEN PRINT "Your heart beats faster..."
950 IF RND(0)>.6 THEN PRINT "Dust and cobwebs are everywhere."
960 IF EV>0 THEN PRINT "There is evil lurking about..."
970 IF EV>10 AND EV<21 THEN PRINT "You hear strange noises about you"
980 IF EV>21 THEN PRINT "There is much evil in the room !!!"
990 IF NOT ((EV>0 AND RND>.75) OR EV>21) THEN RETURN
1000 FOR AI=1 TO MR(PF,ROOM,0)
1010 AD=MR(PF,ROOM,AI) : AE=MM(PF,ROOM,AI) : IF AE<0 AND RND>.5 THEN 1030
1020 NEXT : GOTO 1000
1030 PRINT "Watch out! A ";OBJ$(AD);" ";CRYS$(INT(RND*CRY));" at you!!!"
1040 AE=ABS(AE) : PX=1 : RETURN
1050 '***** LEAVE ****
1060 IF ROOM<0 THEN PRINT "Leave what, you are in a hallway!" : RETURN
1070 IF NOT (AD>0 AND RND>.66) THEN 1100
1080 PRINT "You cannot find the door to the room..."
1090 PRINT "The ";OBJ$(AD);" is after you!!!" : PX=1 : RETURN
1100 PD=1 : IF (ROOM MOD 2)=0 THEN PD=3
1110 ROOM=-((ROOM-1)\2)-1:PRINT"You are back in the hallway"
1120 AD=0 : AE=0 : AI=0 : RETURN
1130 '***** MOVE *****
1140 IF ROOM>0 THEN PRINT "You moved across the room" : RETURN
1150 IF PD=2 AND ROOM=-3 AND PF=1 THEN PRINT "You have walked out of the house":
PX=2:RETURN
1160 IF (PD=0 AND ROOM=-3) OR (PD=2 AND ROOM=-1) THEN PRINT "You're in the middl
e of the hall":ROOM=-2:RETURN
1170 IF PD=0 AND ROOM=-2 THEN PRINT "You're at the end of the hallway":ROOM=-1:R
ETURN
1180 IF PD=2 AND ROOM=-2 THEN PRINT "You're at the front of the hallway":ROOM=-3
:RETURN
1190 IF (PD=0 AND ROOM=-3) OR (PD=2 AND ROOM=-1) THEN PRINT "Ouch! You bumped in
to a wall in the hallway":RETURN
1200 IF (PD=3 AND ABS(ROOM)<=RMR) OR (PD=1 AND ABS(ROOM)<=RML) THEN 1220
1210 PRINT "Thump!  You are facing a wall you bumped into!":RETURN
1220 PRINT "You are standing before a doorway.  Dare you enter?"
1230 IF PD=1 THEN DOOR=(ABS(ROOM)*2) ELSE DOOR=(ABS(ROOM))*2-1
1240 RETURN
1250 '***** VOCAB ****
1260 PRINT "Try the following: ";
1270 FOR I=1 TO NKW : IF I<>1 THEN PRINT ", ";
1280 PRINT KW$(I);: NEXT
1290 PRINT:RETURN
1300 '***** TAKE ****
1310 IF PI=5 THEN PRINT "You have taken enough..." : PX=1 : RETURN
1320 IF PW=NPW THEN PRINT "Take what???" : PX=1 : RETURN
1330 PW=PW+1 : W$=PW$(PW) ' get next word
1340 FOR I=1 TO MR(PF,ROOM,0)
1350 J=MR(PF,ROOM,I) : IF OBJ$(J)=W$ THEN 1380 ' object in room?
1360 NEXT
```

```
1370 PRINT "There is no ";W$;" in the room" : RETURN
1380 PTV=MM(PF,ROOM,I) : IF PTV>100 THEN 1430 ELSE IF PTV>0 THEN 1410
1390 PRINT "You can't take a ";W$ ' this is a no-no
1400 PRINT "It ";CRYS$(INT(CRY*RND));" at you" : PX=1 : RETURN
1410 PWS=PWS+PTV : PRINT "The ";W$;" has a power of";PTV ' weapon power
1420 GOTO 1440
1430 PTS=PTS+(PTV MOD 100) : PRINT "You now have";PTS;"points"  ' treasure
1440 PI=PI+1 : FOR K=1 TO 5 : IF PITM$(K)<>"" THEN NEXT ' find a spot
1450 PITM$(K)=W$ : PTV=MM(PF,ROOM,I)     ' add to player's inventory of items
1460 IF PTV<100 THEN PIM(K)=PTV ' power of item
1470 MM(PF,ROOM,I)=0 : MR(PF,ROOM,I)=0  ' remove it from the room
1480 RETURN
1490 '***** INVEN ****
1500 N=0 : PRINT "You have ";
1510 FOR I=1 TO 5 : IF PITM$(I)="" THEN 1530 ELSE IF N<>0 THEN PRINT ", ";
1520 N=N+1 : ITM$=PITM$(I) : GOSUB 1860
1530 NEXT
1540 IF N=0 THEN PRINT "nothing!" ELSE PRINT
1550 RETURN
1560 '***** USE ****
1570 GOSUB 1770 : IF IT=0 THEN RETURN ELSE IF MI>0 THEN 1590 ' get item
1580 PRINT "You can't fight with "; : GOSUB 1860 : PRINT : RETURN
1590 IF AD=0 THEN PRINT "You just wasted your ";ITM$ : GOTO 1650
1600 IF MI>=AE THEN GOTO 1670 ELSE IF MI<AE/3 THEN GOTO 1710
1610 IF MI<AE/2 THEN PRINT "Try again... you wounded the ";OBJ$(AD)
1620 IF MI<AE THEN PRINT "The ";OBJ$(AD);" is badly hurt!!!"
1630 PRINT "It ";CRYS$(INT(RND*CRY));" and ";ATKS$(INT(RND*ATK));" at you!"
1640 AE=AE-MI
1650 PI=PI-1 : PWS=PWS-MI : MI=0 : PITM$(IT)="" : PIM(IT)=0 : RETURN
1660 ' good triumphs
1670 PRINT "You have won this battle..."
1680 PRINT "The ";OBJ$(AD);" ";WINS$(INT(WIN*RND))
1690 AD=0 : PS=PS+AE : GOTO 1640
1700 ' evil triumphs
1710 PRINT "The ";OBJ$(AD);" has won..."
1720 PRINT "You are ";LOSS$(INT(LOS*RND))
1730 AD=0 : PS=PS-AE : PX=2 : GOTO 1640
1740 '
1750 ' use some item
1760 '
1770 IT=0 : IF PW=NPW THEN PRINT "Huh... use what?" : RETURN
1780 PW=PW+1 : ITM$=PW$(PW) ' get next word (some item)
1790 FOR I=1 TO 5
1800 IF PITM$(I)=ITM$ THEN IT=I : MI=PIM(I) : PIM(I)=0 : RETURN
1810 NEXT
1820 PRINT "You don't have "; : GOSUB 1860 : PRINT : RETURN
1830 '
1840 ' print an item
1850 '
1860 IF INSTR("aeiou",LEFT$(ITM$,1))=0 THEN PRINT "a "; ELSE PRINT "an ";
1870 PRINT ITM$; : RETURN
1880 '
1890 ' set variables when player on floor PF
1900 '
1910 RMS=MF(PF) : RML=INT(RMS/2+.5) : RMR=INT(RMS/2) : DOOR=0 : ROOM=-2
1920 PD=0 : RETURN
1930 '
```

```
1940 ' extract words from player's entry
1950 '
1960 NPW=0 : W$="" : PE$=PE$+" "
1970 WHILE LEN(PE$)>0
1980    C=ASC(PE$) : PE$=MID$(PE$,2)
1990    IF C=32 AND W$<>"" THEN NPW=NPW+1:PW$(NPW)=W$:W$="":IF NPW>14 THEN 2040
2000    IF C>64 AND C<91 THEN C=C+32
2010    IF C>96 AND C<123 THEN W$=W$+CHR$(C)
2020 WEND
2030 RETURN
2040 BEEP : PRINT : PRINT "Couldn't read all ";PE$ : RETURN
2050 '
2060 ' Display contents of manor house; use with a GOTO this line of code
2070 '
2080 CLS : PRINT : PRINT "House number";H;"has";NF;"floors."
2090 FOR I=1 TO NF
2100 PRINT : PRINT " Floor";I;"has";MF(I);"rooms."
2110   FOR J=1 TO MF(I)
2120   PRINT "  Room";J;"has";MR(I,J,0);"objects, with"MM(I,J,7)"points evil,";
2130   PRINT MM(I,J,8) "of treasure, and " MM(I,J,9) " of power"
2140   PRINT TAB(10);
2150      FOR K=1 TO MR(I,J,0)
2160      PRINT USING "& ###_, " ; OBJ$(MR(I,J,K));MM(I,J,K) MOD 100;
2170      NEXT : PRINT
2180   NEXT
2190 NEXT
2200 END
2210 '
2220 ' load keywords, phrases, and items for rooms
2230 '
2240 READ NKW : FOR I=1 TO NKW   : READ KW$(I) : NEXT
2250 READ CRY : FOR I=0 TO CRY-1 : READ CRYS$(I) : NEXT
2260 READ ATK : FOR I=0 TO ATK-1 : READ ATKS$(I) : NEXT
2270 READ WIN : FOR I=0 TO WIN-1 : READ WINS$(I) : NEXT
2280 READ LOS : FOR I=0 TO LOS-1 : READ LOSS$(I) : NEXT
2290 ON ERROR GOTO 2310
2300 FOR OBJ=0 TO 99 : READ OBJ$(OBJ),TYP(OBJ) : NEXT ' read till error
2310 IF ERR<>4 THEN ON ERROR GOTO 0 ELSE RESUME 2350
2320 '
2330 ' set-up manor house H
2340 '
2350 OBJ$(99)="fair damsel" : NF=INT(RND*3)+2  '  2 to 5 floors
2360 GTM=0 : GTP=0 : GTE=0
2370 FOR I=1 TO NF
2380 MF(I)=INT(RND*5)+2 ' from 2 to 6 rooms per floor
2390   FOR J=1 TO MF(I)
2400   MR(I,J,0)=INT(RND*4)+1  ' from 1 to 5 "objects" per room
2410   TE=0 : TP=0 : TM=0
2420     FOR K=1 TO MR(I,J,0)
2430     L=INT(RND*OBJ)          ' which object
2440     MR(I,J,K)=L : M=TYP(L)
2450     IF M=1 THEN N=INT(RND*4+1)*5 : TM=TM+N : MM(I,J,K)=N
2460     IF M=2 THEN N=INT(RND*9+1)*5 : TP=TP+N : MM(I,J,K)=N+100
2470     IF M=3 THEN N=INT(RND*6+1)*5 : TE=TE+N : MM(I,J,K)=-N
2480     NEXT
2490   MM(I,J,7)=TE : MM(I,J,8)=TP : MM(I,J,9)=TM
2500   GTM=GTM+TM : GTP=GTP+TP : GTE=GTE+TE
2510   NEXT
```

```
2520 NEXT
2530 I=INT(RND*NF)+1 : IF I=1 THEN 2530  ' floor (don't allow floor 1)
2540 J=INT(RND*MF(I))+1  ' and room of damsel in distress
2550 K=MR(I,J,0)+1 : MR(I,J,K)=99 : MR(I,J,0)=K
2560 RETURN
2570 '
2580 '  keywords (vocabulary) for player (must be in order shown)
2590 '
2600 DATA 14,look,up,down,left,right,back,enter,leave
2610 DATA    move,vocab,take,inven,use,quit
2620 '
2630 '  assorted phrases
2640 '
2650 DATA 6,shrieks,snarls,spits blood,fumes,glares evilly,cries out
2660 DATA 7,comes at,attacks,approaches,grasps,clutches,grabs for,lunges
2670 DATA 7,dissolves to dust,disappears in a cloud of smoke,withers away
2680 DATA   bursts into flames,changes into a harmless toad,disappears
2690 DATA   turns into a bat and flies away
2700 DATA 5,in their clutches,taken by the evil,mortally wounded
2710 DATA   now in the other world,overcome
2720 '
2730 ' objects for rooms:  weapons, type=1
2740 '
2750 DATA sword,1,       dagger,1,    spear,1
2760 DATA lance,1,       hammer,1,    whip,1
2770 DATA crossbow,1,    knife,1,     razor,1
2780 DATA wand,1,        chain,1,     dart,1
2790 DATA mace,1,        shield,1,    rod,1
2800 '
2810 ' treasures, type=2
2820 '
2830 DATA necklace,2,    silver,2,    urn,2
2840 DATA portrait,2,    statue,2,    gold,2
2850 DATA mirror,2,      scroll,2,    lyre,2
2860 '
2870 ' creatures, type=3
2880 '
2890 DATA witch,3,       goblin,3,    ghost,3
2900 DATA warlock,3,     demon,3,     serpent,3
2910 DATA goblin,3,      lizard,3,    mummy,3
2920 DATA werewolf,3,    ogre,3,      gnome,3
2930 DATA vampire,3,     zombie,3,    wicked elf,3
2940 DATA giant rat,3,   gargoyle,3,  giant snake,3
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 4.1 MANOR - Igor Brimski manor adventure game

Variations

There are many possible additions to our adventure game.

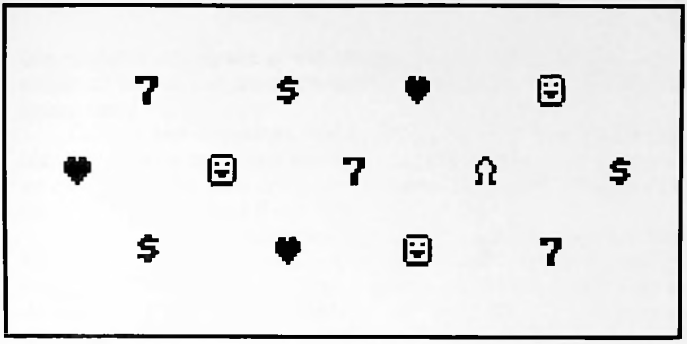
1. Add more objects to the DATA statement tables, located at the end of the program.

2. Create other types of objects. Add some form of magic to increase the power of a player's weapons.

3. Develop some new keywords for doing battle with an adversary. Another possibility is a keyword command that will discard a treasure so you can pick another, more valuable, treasure.

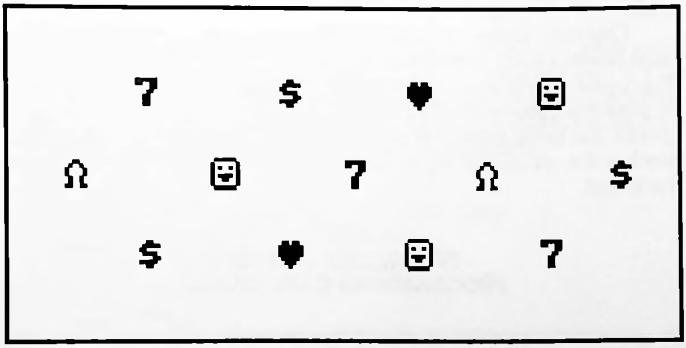
4. Create a more complex house. Rooms could lead to other rooms instead of being entered only from the hallway. (Of course, this is a major change to the program.)

5. Develop a graphics mode version of the program. Display subroutines can be created for an empty room, for each of the objects displayed within the room, and for the hallway, including the appropriate number of doorways. (See the States and Capitals game in Chapter 9 for details on how to create tables of graphics images using the DRAW statement.)



5

Deal 'em



There is an almost universal appeal to card games: Their existence has been traced back to the early Egyptian times, when cards were made of parchment, to when the games were introduced in Europe during the crusades. The year Gutenberg first printed the Bible, he also printed decks of Tarot (fortune) cards.

At times, it seems that there is a countless number of possible games, with bridge and poker leading the pack. A typical "rules according to Hoyle" book might list hundreds of games and variations. Poker is especially littered with variations. No doubt, many variants are invented on the fly at the friendly Friday-night get-together (most likely, by the losers trying to recoup their losses). From a computer-based standpoint, the appeal of the card game is also pervasive. Most, if not all, time-sharing services or data processing centers have at least a blackjack program for the enjoyment of the staff.

Computer-based card games serve a number of purposes. The most obvious is the pure pleasure of playing the game. There is also the opportunity to learn the rules of a new game and develop a skill to play the game well. This is best illustrated by the blackjack programs that teach some sort of optimal playing regimen, how to keep track of the cards, and so on. Programs for teaching bridge can also be found.

TECHNIQUES USED IN PROGRAMMING CARD GAMES

In general, programming card games is fairly simple; but they can also get quite messy, depending on the degree of realism and atten-

tion to detail attempted in the design. Games can be comprised of either all text or text accompanied by graphics to display the cards being used.

Games that are strictly text based are the simplest to develop. Interaction with the computer is accomplished by answering yes or no questions. The state of the game is printed in text form, also, such as "You have a K and 8" or "You just Busted."

Graphics-based games are more exacting in their development. First, the layout of the cards on the screen must be considered. Card images displayed for both the player and the dealer's hands must be developed. For games with many cards per hand, or where more than two players can play, the placement of the cards on the screen can be somewhat involved. When the players place their bets or ask for a card, other images on the screen must not be disturbed.

Fortunately, with the PC, these issues are not much of a problem. The PC's character set has the four-card suit symbols, so designing the card images is easy enough. With the proper use of the LOCATE and PRINT statements, we can design a screen to our liking and control our players' interaction during the game. Before getting into some specific games, let's review a number of items common to all of our card games. These include: How to code a deck of cards as an array, shuffling the deck, creating card images to display on the screen, controlling player interaction in the game, handling multiplayer games, and simulating the computer's play of the cards.

Coding the Card Deck

All card games require at least one deck of cards. The deck is easily coded as a numeric array, such as D(i) where i ranges from 1 to 52 or 0 to 51. As for the values of D(i), we need to encode the suit (clubs, diamonds, spades, or hearts) and the face value. This can be done by simply coding card numbers from 0 to 51 where:

```
Suit = Card number \ 13 ' integer divide
     = 0, 1, 2, or 3 ' suit numbers
```

and

```
Face value = Card number MOD 13 ' modulor operator
           = 0, 1, . . . 11, 12 ' 2, 3, . . . Queen, King
```

Suit numbers from 0 to 3 can be converted later on when displaying the cards to the appropriate PC character code by adding a 3 to each number. The PC character codes 3, 4, 5, and 6 are the heart, diamond, club, and spade, respectively. For instance, when the character functions CHR$(3) is printed on the screen, it displays a heart symbol; CHR$(4) represents a diamond, and so forth.

Another card number coding approach is to take the suit number, multiply it by 100, and add the face-value number. Then the operations to determine the suit and face value become:

Suit = Card number \ 100
Face value = Card number MOD 100

Although there is no difference in the two techniques, the latter allows for easier program debugging. Consider looking at a card number that is 305, which is the five of clubs, versus the card number 37, which is . . . well, first we divide by . . . , and so on.

To set up a deck, we need only two FOR / NEXT loops: one for the suit value and the second for the face value. Since we have multideck games, we can add a third FOR / NEXT loop for the number of decks. For the programs that follow, the following subroutine will be used, where ND stands for the number of decks to be initialized.

```
5140 '
5150 ' set up deck
5160 '
5170 NC=ND*52-1 : DIM D(NC) : P=-1 ' number cards, deck array and pointer
5180 FOR I=1 TO ND : FOR J=1 TO 4  ' for each deck and suit
5190   FOR K=2 TO 14 : P=P+1 : D(P)=J*100+K : NEXT ' set card numbers
5200 NEXT : NEXT
5210 P=ND*52 : RETURN ' set pointer to force shuffle
```

Note that the suit values range from 1 to 4 and the face values range from 2 to 14. To convert this suit number to its character code, we add a 2. In using the face values, the values of 2 through 10 stand for the number cards, and the 11, 12, 13, and 14 stand for the jack, queen, king, and ace.

Shuffling the Deck

Now that we have an array of card numbers (our deck), it's time to shuffle them. One approach is to exchange pairs of randomly se-

lected elements from the array with each other. For example, if the variable NC is the number of cards in the deck, then the following statement exchanges two random elements:

```
SWAP D(INT(RND*NC)),D(INT(RND*NC))
```

where the expression INT(RND*NC) evaluates to a random index, 0 to NC-1, in the card array. Note that this is another reason for indexing the deck from zero instead of one. A drawback to the above is its unpredictable nature: Array elements may be selected more than once and, occasionally, the same element will be exchanged with itself. The RND function is also fairly slow, so a shuffle for a large number of decks can take a while.

We can improve upon this with an approach that exchanges every card in succession with one randomly selected from the cards that have not been shuffled. A FOR / NEXT loop can step through each card, 0 to NC-1, with the index I, SWAPping D(I) with a card selected from the values in the range of array elements from I+1 to NC-1 (that is, those elements beyond the index I). This ensures that every card is exchanged at least once. It also uses the RND function just once per exchange. The subroutine to shuffle is as follows:

```
5220 '
5230 ' shuffle the cards
5240 '
5250 T$="Excuse me while I shuffle the deck..." : GOSUB 5120
5260 FOR I=1 TO 3 : FOR P=0 TO NC  ' three passes over deck
5270 SWAP D(P),D(INT(RND*NC))      ' exchange each card with a random one
5280 NEXT : NEXT : P=-1            ' set deck pointer when done
5290 T$="":GOSUB 5120:R$=" ':T$="All done. Press <SPACE> to play":GOSUB 5030
5300 RETURN
```

The T$ and R$ variables in line 5290 are used by the subroutines called in those lines to print a message on the screen for the player and to get his response in order to continue. They are explained in more detail in a later section.

Card Image Graphics

Now that we have our shuffled deck available for play, we need to consider how to display the cards. Numerous formats are possible, from simple-lined boxes, with the card value printed in the interior, to a full representation of the card, complete with suit symbols and even a face for the king, queen, and jack.

The card images in the games that follow will all be based on character mode graphics. (Representing the faces of a king, queen, or jack will not be attempted.) To show a numeric face value with a full complement of suit symbols (that is, all ten hearts for the 10 of hearts) will require a fairly large image plus some coding scheme for placement of the suit symbols. We will opt for a card image that works as follows.

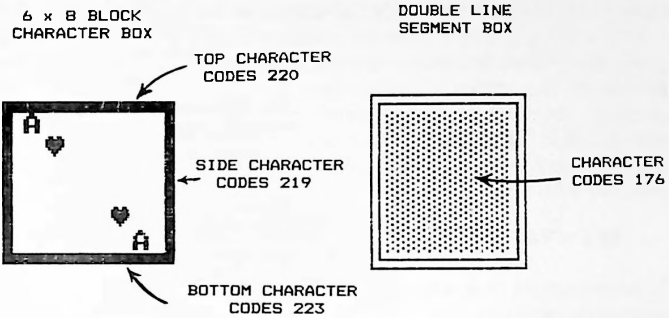
Each card is displayed in a box whose dimensions are 6 characters down and 8 characters across. The face number of each card is displayed in the upper-left and lower-right corners. The suit symbol is displayed below and above these values. The 6- by 8-space requirement is a good intermediate size that allows placement of quite a few cards on the screen without crowding the display. The following subroutine is used.

```
5310 '
5320 '   Face of 6x8 card
5330 '
5340 COLOR ABS((D\100)<3)*4,7
5350 F1$=MID$("*23456789JQKA",D MOD 100,1):F2$=F1$
5360 IF (D MOD 100)<>10 THEN F1$=F1$+" ":F2$=" "+F2$ ELSE F1$="10":F2$="10"
5370 LOCATE R+0,C:PRINT CHR$(218);STRING$(6,196);CHR$(191);
5380 LOCATE R+1,C:PRINT CHR$(179);F1$+"    ";CHR$(179);
5390 LOCATE R+2,C:PRINT CHR$(179);" ";CHR$(D\100+2);"    ";CHR$(179);
5400 LOCATE R+3,C:PRINT CHR$(179);"    ";CHR$(D\100+2);" ";CHR$(179);
5410 LOCATE R+4,C:PRINT CHR$(179);"    "+F2$;CHR$(179);
5420 LOCATE R+5,C:PRINT CHR$(192);STRING$(6,196);CHR$(217);
5430 COLOR 7,0:RETURN
```

The subroutine takes as input the card number, D, and the row and column position of the card, R and C. Using the card number, the subroutine first determines the color of the suit value that will be used for the card: Red on white for hearts and diamonds, and black on white for clubs and spades. The suit number is also used to calculate the PC's character code for the suit symbol, 3 to 6, by adding 2 to the suit symbol in line 5390 when the center of the card is printed.

The face value is then converted to its numeric form for numbers from 2 to 10, and to an A, K, Q, and J for the face cards. A letter "O" is used in the 10 instead of a zero purely for appearance. The top and bottom lines in the card are set in the variables F1$ and F2$. Each line in the card is then printed out in succession. The character codes for a single-line box are included to create a border around the card. See Figure 2.1 on page 28 for an illustration of these character codes. A LOCATE statement, using the R and C (row and column) position

FIGURE 5.1. Front and back of a 6-row by 8-column playing card.



variables, is made at the appropriate position on the screen for printing each line.

While we are at it, we will code an image for the back of the card. This is useful for showing cards as discarded, or whenever you want to show a card on the screen before it is turned over to reveal its face.

```
5440 '
5450 '    Back of 6x8 card
5460 '
5470 COLOR 0,7:LOCATE R,C:PRINT CHR$(201);STRING$(6,205);CHR$(187)
5480 FOR K=R+1 TO R+4
5490 LOCATE K,C:PRINT CHR$(186);STRING$(6,176);CHR$(186)
5500 NEXT
5510 LOCATE R+5,C:PRINT CHR$(200);STRING$(6,205);CHR$(188);
5520 COLOR 7,0:RETURN
```

This subroutine operates by printing a double-lined box filled in with the shading character, number 176, which actually resembles the back of some makes of cards. The R and C variables once again serve to indicate where to display the image. See Figure 5.1 for an illustration of the front and back of a typical playing card displayed via the subroutines described in this section.

Player Input and Ouput

Control of the player interaction during the game is necessary in order to prevent disturbing the screen layout in the event that the player makes an erroneous entry. BASIC's ubiquitous "Redo from

start" message is the major source of the irritant, plus BASIC's enabling you to move the cursor anywhere on the screen.

The solution to this potential problem is handled as follows. All prompts between the player and computer will be transacted at the bottom of the screen. Player responses may be either single-character or multiple-character values. For multicharacter replies, the LINE INPUT statement will be used with the user's reply stored in the variable A$. For numeric replies, the A$ is converted to a number with the VAL function, such as for the betting process:

```
BET = VAL(A$)
```

This overcomes user attempts to enter nonnumeric data when an amount is required. For instance, if A$="YES", indicating that the user entered yes to the request for a bet amount, then VAL(A$) will be zero. The program checks for bets less than or equal to zero, which filters out this erroneous entry.

One area where the LINE INPUT statement does not help is in nullifying the use of the cursor control keys. While responding to a query, the user has free rein to move the cursor anywhere on the screen, potentially disturbing the layout of the screen. Solving this problem requires the use of a subroutine that processes all user replies on a character-by-character basis with the INKEY$ function, disallowing any cursor movement outside the area reserved for data entry. This also means that any editing that the user makes in his replies, such as using the backspace, Ins, Del, keys, and so on, must also be processed by the subroutine. This is left for the ambitious user to incorporate into the programs.

For single-character user replies, we can filter out all but the acceptable replies. Our subroutine will use a WHILE / WEND loop with the INKEY$ function to read the single character. To edit the character a variable R$ will be set to the valid user replies. We can then easily check that the single character keyed in by the user is acceptable by using the INSTR function. After an invalid entry is made a BEEP is sounded and another character is retrieved. Otherwise the subroutine erases the prompt and returns to the main program.

Each character must be shifted to upper case before it can be used. For instance, to get a yes/no reply such as Y or N, the R$ value

would be set to "YN" prior to calling the subroutine. In this case, only a Y or N will be accepted by the program—the user simply presses either the Y or N key. When the R$ variable is null, it indicates that a multicharacter input is to be made using the LINE INPUT statement.

The text requesting the user replies in the card games will all be displayed on row 23 of the screen. The prompting text is stored in string variable T$. When displayed, the prompting text is centered on row 23 of the screen. The complete subroutine is shown in lines 5000 to 5080.

```
5000 '
5010 ' get player responses
5020 '
5030 LOCATE 23,40-LEN(T$)/2 : PRINT T$;
5040 IF R$="" THEN LINE INPUT;"",A$ : GOTO 5080 ' multi-char reply
5050 C$="" : WHILE C$="" : C$=INKEY$ : WEND  ' or single character
5060 IF C$>"Z" THEN C$=CHR$(ASC(C$)-32)      ' shift to upper case
5070 IF INSTR(R$,C$)=0 THEN BEEP : GOTO 5050 ' validate character
5080 LOCATE 23,1 : PRINT SPC(79); : RETURN
5090 '
5100 ' display a message
5110 '
5120 LOCATE 21,1 : PRINT SPC(79);
5130 LOCATE 21,40-LEN(T$)/2 : PRINT T$; : RETURN
```

A number of messages are used in the programs to indicate the status of play, or to reinforce the prompting used with the procedures discussed in the previous paragraphs. An example is the message that the deck is being shuffled, or that the game is over because everyone is broke, and so on. These messages are treated in the same way as the prompts.

The subroutine at 5120 is called each time such a message is to be displayed. The variable T$ is set to the text of the message. A GOSUB is then made, and the message is printed, centered on row 21. To erase a message displayed by this subroutine, set T$ to the null string and call the subroutine. The previous message is then replaced by blanks.

Multiplayer Games

As in any game, the computer can play the role of croupier (referee, umpire, and so on), a fellow player, or an opponent. Thus, programs are often developed for two players: the human user and the com-

puter—but why stop there? Almost all card games allow three or more players.

The programming methodology to accommodate additional human players means that a loop of some sort must be used to process each player in succession, with the status of each player stored in one or more arrays. For example, each player's current bankroll, his initial capital, and the amount of his current bet must be remembered. In the card games to follow, there are two multiplayer games: two-card monte (a type of poker) and blackjack. The processing of each player's input is done with the following programming conventions:

- The variable NP contains the number of players playing the game minus one. Thus, for a three-player game, NP will be 2. Numbering from zero has benefits in calculating the screen position of various player-specific messages (a player's bankroll, the bet, the player's name, and so on).
- The array N$(i) contains the name of each player. It allows a degree of "personalization" by allowing the program to query each of the players by name. The routine concatenates the name in N$ with the appropriate message to create T$, which is then printed on the screen by the prompting subroutine starting at line 5000. The first letter in each player's name is also changed to upper case to ensure that it is correctly displayed.
- The two arrays BR(i) and BP(i) contain each player's current bankroll, the money each has to play with, and each player's current bet for this hand.

Some minor problems can arise in multiplayer games. For instance, if all but one player bankrupts, do you continue the game or not? This usually depends on the type of game. Of course, the computer can always "loan" the bankrupt player some money to continue. But suppose there is a tie among the winners of a hand? The program must split the pot between the winning players. A potential problem in the games with a variable number of cards, such as blackjack, is positioning each player's cards so that they do not overlap the other players' cards.

Last, for the games where down cards are dealt for each player, each player must be able to see his cards but without the other players seeing them. For this reason, the game of five-card draw has been limited to a single human player versus the computer. Turning

over hands of five cards for each player to view would become confusing, although it is certainly possible.

In any event, more details on handling multiple players in these games can be found in the descriptions of each game to follow.

The Computer Plays

Programming for the computer's play in card games involves a few special considerations. As each player takes a turn, there is usually a pause to consider the bet, whether or not to play, and so on. When it's the computer's turn, it makes these decisions almost instantaneously. Thus, to imitate a humanlike game flow, strategically placed pauses can be useful.

Consider using a time delay created by a FOR / NEXT loop when it's the computer's turn to bet. This will make the computer count from, for example, one to one thousand. You can adjust the upper limit for a pause to give the appearance of the computer pondering its bet. This can also be used to keep the screen display from changing too rapidly as cards are dealt.

The actual calculations for playing or not, that is, whether to bet, raise, or fold, also bear commenting. First and foremost, the code used must not consider any data that would not be available to human players. Thus, the player's down cards should not come into the calculations, although the value of the player's face up cards certainly should.

Second, there are basically two areas of decision making. If the current bet is zero, then the computer is being asked to open, or begin the betting. If the bet is *not* zero, then the computer must either bet that same amount, a "call," or decline to play by betting zero, which is a "fold." The computer may also increase the bet ("raise") if it has the money to do so.

In all cases, the computer's reply can be based on the value of its own hand, information about the exposed cards (if any), its bankroll, the player's bankrolls, the current bet (if any), and whether it wants to bluff or not. Bluffing would be based on some random value from the random number function, RND. This list of items could also be expanded upon by the computer keeping tabs on the betting style of the players. This can become fairly involved and will be left to the interested reader to consider.

THE GAMES

There are four card games developed in this chapter:

1. High Card: A high/low card-betting game between the computer and player. This game introduces all of the concepts discussed above for a very simple but fun game.

2. Five-Card Draw: This is a complete implementation of the most fundamental of all poker games. The player and the computer may bet, raise, and fold. A draw of one, two, or three cards is allowed after the betting is completed.

3. Two-Card Monte: From one to four players may play along with the computer, dubbed the "PC Kid," in a two-card poker game. Players may also bet, raise, or fold. There is no draw. This game introduces how multiplayer card games are programmed.

4. Blackjack: This game is standard among computer-based card games. This implementation allows one to four players, with the computer acting as the dealer.

These games are the longest programs in the book (other than the backgammon game in Chapter 7) in terms of the number of lines of code. On the other hand, the common features in each game, discussed in the opening sections of this chapter, comprise about one-fourth of each program. So, keying in one game means you have also completed a good portion of the others.

High Card

High Card is the simplest and most basic of our card games: You and the computer are each dealt a single card, which you must either play with and bet, or fold. Each of you has the same initial bankroll and must ante some fixed amount of money before the two cards are dealt. A player who folds forfeits his ante to the other player—so waiting for a high card will slowly eat away at your bankroll. See Figure 5.2 for an example of the game being played.

After the two cards are dealt—yours face up, the computer's face down—the pot is automatically updated with the ante. Interacting with the game is as follows: You press a P to play or an F to fold.

FIGURE 5.2. High-card game example.

```
HIGH CARD

POT 202

PLAYER'S CARD                    COMPUTER'S CARD

K♥                               6♦
   ♥                                ♦
      K                                6

BANKROLL                         BANKROLL

5398                             4400

Player wins

Press <SPACE> to pay off bets...
```

To play means you want to bet some amount in addition to the ante already taken. After your bet, the computer can either call, meaning it matches your bet, or it passes, meaning it folds and forfeits its ante. If you had decided to initially fold (not play), then you would have lost your ante to the computer. If a bet and call are made, then the cards are turned over and the payoff is made. The high card wins, unless the cards are alike, which is a draw. The computer then takes a turn playing in the same manner.

The main variables in the game are the money amounts: PB, CB, POT, and B, variables for the player's bankroll, the computer's bankroll, the money in the pot, and the current bet. Note the symmetry in the dialogue for making the bets. Note also the extensive use of the subroutines at 5030 and 5120 for carrying on the dialogue. This is the code that was discussed in the opening sections of this chapter. The complete listing of the one-card game is shown in Program 5.1.

```
0 ' HCARD  High card betting game
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 ' Ante every hand and either play and bet or fold.  High card wins.
50 '
60 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
70 DEFINT A-Z : RANDOMIZE VAL(RIGHT$(TIME$,2))
80 PRINT"HCARD - High Card Betting Game" : PRINT
90 LINE INPUT "What is the ante amount (from 1 to 500)? ",N$
100 A=VAL(N$) : IF A<1 OR A>500 THEN BEEP : GOTO 90
110 PRINT "What are the bankrolls (from";A*25;"to 25000)? ";:LINE INPUT N$
120 B=VAL(N$) : IF B<A*25 OR B>25000 THEN BEEP : GOTO 110
130 POT=0 : PB=B : CB=B  ' initialize pot, player and computer bankrolls
140 ND=1  : GOSUB 5170   ' set the D array, deck, and P, card pointer
150 '
160 ' format screen
170 '
180 CLS
190 COLOR 0,7 : LOCATE 1,32 : PRINT " H I G H   C A R D "
200 LOCATE 3,33 : PRINT "   POT          " : COLOR 7,0
210 LOCATE 5,12 : PRINT "PLAYER'S CARD";
220 LOCATE 5,52 : PRINT "COMPUTER'S CARD";
230 LOCATE 14,15: PRINT "BANKROLL";
240 LOCATE 14,55: PRINT "BANKROLL";
250 '
260 ' player has cards
270 '
280 IF P>NC-2 THEN GOSUB 5250 ' shuffle if less than 2 cards
290 GOSUB 770                 ' deal the cards
300 R$="PF" : T$="Press 'P' to Play or 'F' to Fold" : GOSUB 5030
310 IF C$="F" THEN CB=CB+POT : GOTO 390 ' player folds
320 IF C$<>"P" THEN BEEP : GOTO 300
330 R$="" : T$="Enter your bet: " : GOSUB 5030  ' ask for player's bet
340 B=VAL(A$):IF B<1 OR B>PB OR B>CB THEN BEEP:GOTO 330 ' valid?
350 PB=PB-B : POT=POT+B : GOSUB 860 : GOSUB 980 ' computer decides
360 IF CP=0 THEN T$="Computer passes" : GOSUB 5120 : PB=PB+POT : GOTO 390
370 T$="Computer decides to call" : GOSUB 5120
380 CB=CB-B : POT=POT+B : GOSUB 860 : GOSUB 680  ' compare the cards
390 POT=0 : R$=" " : T$="Press <SPACE> to pay off bets..." : GOSUB 5030
400 GOSUB 860 : GOSUB 930 : GOSUB 1100 : IF BROKE THEN 620  ' anyone busted?
410 T$="":GOSUB 5120:R$=" ":T$="Press <SPACE> for computer's turn":GOSUB 5030
420 '
430 '  computer's turn
440 '
450 IF P>NC-2 THEN GOSUB 5250   ' time to shuffle?
460 B=0 : GOSUB 770 : GOSUB 980 ' deal cards and let computer pass or fold
470 IF CP=0 THEN T$="Computer folds." : GOSUB 5120 : PB=PB+POT : GOTO 550
480 T$="Computer decides to play and bets $"+STR$(B) : GOSUB 5120
490 CB=CB-B : POT=POT+B : GOSUB 860
500 R$="CP" : T$="Press 'C' to Call or 'P' to Pass": GOSUB 5030
510 IF C$="P" THEN CB=CB+POT : T$="" : GOSUB 5120 : GOTO 550
520 IF C$<>"C" THEN BEEP : GOTO 500
530 T$="So you want to call..." : GOSUB 5120
540 PB=PB-B : POT=POT+B : GOSUB 860 : GOSUB 680  ' pots right, show down
550 POT=0:R$=" ":T$="Press <SPACE> to pay off bets...":GOSUB 5030
560 GOSUB 860 : GOSUB 930 : GOSUB 1100 : IF BROKE THEN 620  ' anyone busted?
570 T$="":GOSUB 5120:R$=" ":T$="Press <SPACE> for player's turn":GOSUB 5030
```

```
580 GOTO 280 ' next hand
590 '
600 ' end game, or play some more
610 '
620 R$="YN" : T$="Want to play another game (Y or N)?" : GOSUB 5030
630 IF C$="Y" THEN CLS : ERASE D : GOTO 80 ' clear D for new deck
640 IF C$<>"N" THEN BEEP : GOTO 620 ELSE C$=CHR$(27) : GOTO 9050
650 '
660 ' compare two cards
670 '
680 R$=" ":T$="Press <SPACE> to compare the two cards...":GOSUB 5030
690 R=7:C=55:GOSUB 5340 ' turn over computer's card
700 IF PC=CC THEN PB=PB+POT/2 : CB=CB+POT/2 : T$="Its a draw!  Nobody wins..."
710 IF PC>CC THEN PB=PB+POT : T$="Player wins"
720 IF CC>PC THEN CB=CB+POT : T$="Computer wins"
730 GOSUB 5120 : POT=0 : RETURN
740 '
750 ' deal two cards
760 '
770 POT=2*A : PB=PB-A : CB=CB-A : GOSUB 860 ' update pot with antes
780 P=P+1 : D=D(P) : R=7:C=15 : GOSUB 5340  ' show player's card up
790 PC=D MOD 100  ' player's card value
800 P=P+1 : D=D(P) : R=7:C=55 : GOSUB 5470  ' deal computer's down
810 CC=D MOD 100  ' computer's card value
820 RETURN
830 '
840 ' display all money
850 '
860 LOCATE 16,18 : PRINT USING "######"; PB
870 LOCATE 16,58 : PRINT USING "######"; CB
880 COLOR 0,7 : LOCATE  3,40 : PRINT USING "######"; POT : COLOR 7,0
890 RETURN
900 '
910 ' clear card area
920 '
930 FOR R=7 TO 12 : LOCATE R,15 : PRINT SPC(48); : NEXT
940 RETURN
950 '
960 ' computer ponders a fold, pass or bet
970 '
980 CP=0 : IF B=0 THEN 1030
990 IF CC>10 THEN CP=1:RETURN
1000 IF CC>7 AND (B<CB/2 OR RND>.5) THEN CP=1:RETURN
1010 IF CC>3 AND (B<CB/5 OR RND>.9) THEN CP=1:RETURN
1020 RETURN
1030 IF CC>10 THEN CP=1:B=((CB-10)\30)*10+10:GOTO 1060
1040 IF CC>7 AND (PB/CB<1 OR RND>.8) THEN CP=1:B=((CB-10)\40)*10+10:GOTO 1060
1050 IF CC>3 AND (PB/CB<.5 OR RND>.9) THEN CP=1:B=((CB-10)\50)*10+10
1060 IF B<PB THEN RETURN ELSE B=PB : RETURN
1070 '
1080 '  anyone bankrupt?
1090 '
1100 BROKE=0 : IF PB>A AND CB>A THEN RETURN
1110 IF CB=0 THEN T$="You just bankrupt the computer!!"
1120 IF CB<=A THEN T$="Computer is busted!"
1130 IF PB=0 THEN T$="Computer has bankrupt you!!"
1140 IF PB<=A THEN T$="You are busted!  Sorry..."
1150 GOSUB 5120 : BROKE=-1 : RETURN
```

```
5000 '
5010 ' get player responses
5020 '
5030 LOCATE 23,40-LEN(T$)/2 : PRINT T$;
5040 IF R$="" THEN LINE INPUT;"",A$ : GOTO 5080 ' multi-char reply
5050 C$="" : WHILE C$="" : C$=INKEY$ : WEND  ' or single character
5060 IF C$>"Z" THEN C$=CHR$(ASC(C$)-32)      ' shift to upper case
5070 IF INSTR(R$,C$)=0 THEN BEEP : GOTO 5050 ' validate character
5080 LOCATE 23,1 : PRINT SPC(79); : RETURN
5090 '
5100 ' display a message
5110 '
5120 LOCATE 21,1 : PRINT SPC(79);
5130 LOCATE 21,40-LEN(T$)/2 : PRINT T$; : RETURN
5140 '
5150 ' set up deck
5160 '
5170 NC=ND*52-1 : DIM D(NC) : P=-1 ' number cards, deck array and pointer
5180 FOR I=1 TO ND : FOR J=1 TO 4  ' for each deck and suit
5190   FOR K=2 TO 14 : P=P+1 : D(P)=J*100+K : NEXT ' set card numbers
5200 NEXT : NEXT
5210 P=ND*52 : RETURN ' set pointer to force shuffle
5220 '
5230 ' shuffle the cards
5240 '
5250 T$="Excuse me while I shuffle the deck..." : GOSUB 5120
5260 FOR I=1 TO 3 : FOR P=0 TO NC  ' three passes over deck
5270 SWAP D(P),D(INT(RND*NC))      ' exchange each card with a random one
5280 NEXT : NEXT : P=-1        ' set deck pointer when done
5290 T$="":GOSUB 5120:R$=" ":T$="All done. Press <SPACE> to play":GOSUB 5030
5300 RETURN
5310 '
5320 '   Face of 6x8 card
5330 '
5340 COLOR ABS((D\100)<3)*4,7
5350 F1$=MID$("*234567891JQKA",D MOD 100,1):F2$=F1$
5360 IF (D MOD 100)<>10 THEN F1$=F1$+" ":F2$=" "+F2$ ELSE F1$="10":F2$="10"
5370 LOCATE R+0,C:PRINT CHR$(218);STRING$(6,196);CHR$(191);
5380 LOCATE R+1,C:PRINT CHR$(179);F1$+"    ";CHR$(179);
5390 LOCATE R+2,C:PRINT CHR$(179);" ";CHR$(D\100+2);"    ";CHR$(179);
5400 LOCATE R+3,C:PRINT CHR$(179);"    ";CHR$(D\100+2);" ";CHR$(179);
5410 LOCATE R+4,C:PRINT CHR$(179);"    "+F2$;CHR$(179);
5420 LOCATE R+5,C:PRINT CHR$(192);STRING$(6,196);CHR$(217);
5430 COLOR 7,0:RETURN
5440 '
5450 '   Back of 6x8 card
5460 '
5470 COLOR 0,7:LOCATE R,C:PRINT CHR$(201);STRING$(6,205);CHR$(187)
5480 FOR K=R+1 TO R+4
5490 LOCATE K,C:PRINT CHR$(186);STRING$(6,176);CHR$(186)
5500 NEXT
5510 LOCATE R+5,C:PRINT CHR$(200);STRING$(6,205);CHR$(188);
5520 COLOR 7,0:RETURN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

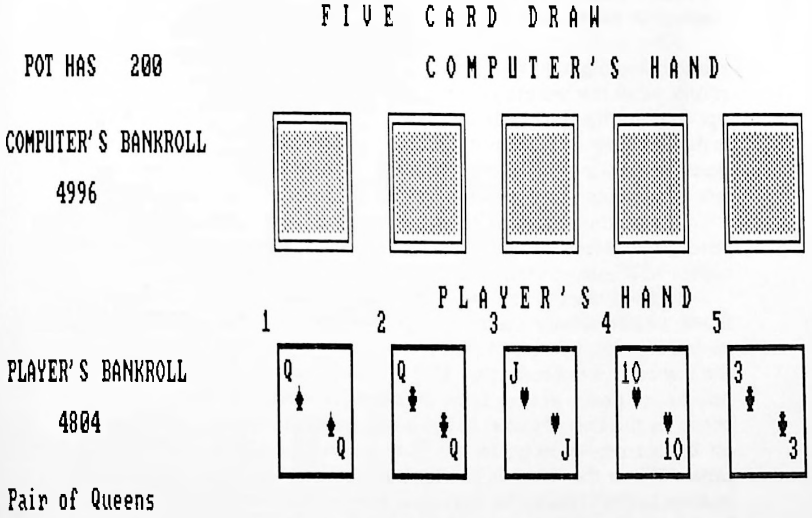
Program 5.1 HCARD - High card poker game

Five-Card Draw

Five-card draw is the classic Friday-night poker game. You and the computer are each dealt five cards, and then you may bet. After all bets and raises are made, you take turns drawing one, two, or three cards. You then bet again, based on your new hand. The player with the highest hand wins. See Figure 5.3 for an illustration of the game being played.

The overall flow in playing this game is similar to the high-card game listed earlier. The difference is in your ability to raise a bet after someone else makes his bet. Thus, if you bet $100 on your hand, the computer may fold, see your $100, or see your $100 and raise you some more money. If a raise is made, then you must bet again and call the raise, fold, or raise again. This continues until both bets are equal, meaning the "pot's right."

FIGURE 5.3. Five-card draw poker card game example.



Thus, the queries posed by this program are prompting you to enter a bet amount instead of the P to pass or C to call as in the high-card game. The computer then checks if you have enough money for such a bet, and whether the bet is equal to the current bet for a "call" or if it is greater than the current bet for a "raise." You cannot make a bet less than the current bet amount except for a zero, which means you fold.

This betting sequence is called the first round of betting. When the first round is concluded, you and the computer may draw from one to three cards. To make your draw, enter one, two, or three numbers in the range of 1 to 5 to indicate which cards in your hand you want to discard. The numbers 1 to 5 are displayed on the screen above the upper-left corner of each card in your hand.

For example, you might enter 45 to discard the fourth and fifth cards in your hand. At most, you may enter three digits. Thus, an entry of 123 indicates that the first three cards are to be replaced. The computer validates your entry, turns over the discards, and then deals your new cards face up.

After both you and the computer draw your cards, a second round of betting follows. This is done in a manner identical to the first round. After the betting is concluded, the computer's hand is turned up, a comparison is made, and the winner is paid off. The next hand is then played. At this point, you may enter either Esc to exit from the game or a space to continue with a new hand of five cards. If either you or the computer goes broke, the game ends automatically.

The data variables used in this program are similar to the high-card game. Money values are PB, CB, and POT for the player, the computer's bets, and the money in the pot, respectively.

To keep track of the two hands, a two-dimensional array is defined, C(2,5), where the first index is 1 or 2 for the computer's or player's cards, which are stored by the second index. Thus, C(1,1) is the first card of the computer, C(1,2) the second, and so on. When the hands are dealt, entries from D, the deck array, are set into the C array for the two players. At the same time, the cards are displayed on the screen—face up for the player, and face down for the computer. When the draw is made, the numbers entered are actually indices to the C array for the cards being replaced. New card numbers from D are then inserted in the array.

The C array is also used by a subroutine to process each hand to determine the hand's numerical value. A description of the value is also determined, such as "3 of a kind" or "High card," and the cards

that should be discarded by the computer. The subroutine is given a pointer that is either 1 or 2 for the hand that is to be evaluated in C.

This card evaluation subroutine, which starts at line 1620, is the bulk of our program in both code and complexity. The general approach is to evaluate the nine types of hands that are possible, whether or not the cards form that hand, the high card for the hand, and the discards that should be used if that hand is played.

The complexity and difficulty in the subroutine lies in the fact that the five cards can be viewed in different ways, depending on how the hand is played. For example, a hand with one pair could also be a potential flush (all the same suit) and a potential straight (numeric sequence). Consider the hand:

- 6 of clubs/7 of clubs/8 of clubs/9 of clubs/9 of spades

Do you discard the 9 of spades in the hopes of getting a flush or a straight, or just dump the 6, 7, and 8, hoping to improve the pair of 9's? This is an extreme example, but illustrates some of the possibilities.

The overall approach of the subroutine is to set a number of variables, CV12, CV23, and so on, and CS12, CS23, and so on, that are the results of comparing each pair of card values and card suits. To do so also requires that the hand be sorted by card value and suit. The results of this sort are saved in the arrays CS, CV. Pointers from the sorted sequence back to the sequence that was dealt are stored in CSP and CVP—this information will be needed when the discards are to be determined.

With the CV* and CS* variables, each type of hand is evaluated. This consists of looking for each of the nine possible poker hands: These are listed below along with the value that is assigned to them:

9. High straight flush—in sequence and same suit, 900
8. Four of a kind—four cards alike, 800
7. Full house—three of a kind, and one pair, 700
6. Flush—all five same suit, 600
5. Straight—all five in sequence, 500
4. Three of a kind—three cards alike, 400
3. Two pair—two sets of two like cards, 300
2. One pair—two cards alike, 200
1. High card—highest card in hand, 100

The overall value of the hand is the 100-to-900 type of hand value listed above, plus the high card in the hand. Thus, a 9-high straight has a value of 509, a pair of 3's is 103, and so on.

When the subroutine has completed its evaluation of the hand, it returns the value of the hand in the variable VH, a textual description of this value in CV$, and the digits of the cards to discard, in DC$. The discard numbers are coded in the same manner as when the player enters his discards. Thus, the same code can process both the player and the computer's discards.

The rest of our draw poker program is relatively simple. When the computer decides to bet or raise, it considers the value of its hand and the current bet amount compared to its hand. For example, if a bet is a relatively small percentage of the computer's bankroll, then the computer tends to be more "aggressive" in its betting. Note also that the initially determined discards for the computer may be changed in certain cases. For example, the computer may "stand pat" occasionally with a two-pair hand. This gives the computer a further degree of realism in play. The listing of the Draw game is found in Program 5.2.

```
0 ' DRAW   5 card draw poker
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 ' PC plays poker: bet, draw 1 to 3 cards, and bet again.  Best hand wins!
50 '
60 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
70 DEFINT A-Z : DIM C(2,5) : RANDOMIZE VAL(RIGHT$(TIME$,2))
80 PRINT"DRAW - Five Card Draw Poker" : PRINT
90 LINE INPUT "What is the ante amount (from 1 to 500)? ",N$
100 A=VAL(N$) : IF A<1 OR A>500 THEN BEEP : GOTO 90
110 PRINT "What are the bankrolls (from";A*25;"to 25000)? ";:LINE INPUT N$
120 B=VAL(N$) : IF B<A*25 OR B>25000 THEN BEEP : GOTO 110
130 POT=0 : PB=B : CB=B ' initialize pot and each player's money
140 ND=1 : GOSUB 5170   ' set up one deck
150 '
160 ' format screen
170 '
180 CLS : COLOR 0,7 : LOCATE 1,30 : PRINT " F I V E   C A R D   D R A W "
190 LOCATE 3,1  : PRINT "  POT HAS          " : COLOR 7,0
200 LOCATE 3,40 : PRINT " C O M P U T E R ' S   H A N D ";
210 LOCATE 12,41: PRINT " P L A Y E R ' S   H A N D ";
220 LOCATE 6,1  : PRINT "COMPUTER'S BANKROLL";
230 LOCATE 15,1 : PRINT "PLAYER'S BANKROLL"; : GOSUB 1420
240 LOCATE 13,24: FOR I=1 TO 5 : PRINT I;SPC(8); : NEXT ' numbers for draw
250 '
260 ' player plays a hand
270 '
280 IF P>NC-16 THEN GOSUB 5250
290 GOSUB 1180 : GOSUB 770 : IF PBET<0 OR CBET<0 THEN 340
300 R$=" ":T$="Press <SPACE> to draw your cards...": GOSUB 5030
```

```
310 GOSUB 1010 : GOSUB 2290
320 R$=" ":T$="Press <SPACE> for final round of betting": GOSUB 5030
330 T$="" : GOSUB 5120 : GOSUB 770 : GOSUB 1480
340 GOSUB 540: IF BROKE THEN 480
350 '
360 ' computer plays a hand
370 '
380 IF P>NC-16 THEN GOSUB 5250
390 GOSUB 1180 : GOSUB 760 : IF PBET<0 OR CBET<0 THEN 440
400 R$=" ":T$="Press <SPACE> for computer's draw": GOSUB 5030: GOSUB 2290
410 R$=" ":T$="Press <SPACE> to draw your cards...": GOSUB 5030: GOSUB 1010
420 R$=" ":T$="Press <SPACE> for final round of betting": GOSUB 5030
430 T$="" : GOSUB 5120 : GOSUB 760
440 GOSUB 540: IF NOT BROKE THEN 280
450 '
460 ' end of game
470 '
480 R$="YN":T$="Want to play another game (Y or N)?":GOSUB 5030
490 IF C$="Y" THEN C$=" " : GOTO 9060 ' restart game
500 IF C$="N" THEN C$=CHR$(27): GOTO 9050 ' exit game
510 '
520 ' end of hand wrap-up
530 '
540 IF PBET<0 THEN CB=CB+POT:POT=0:GOTO 630 ' did player fold?
550 IF CBET<0 THEN PB=PB+POT:POT=0:GOTO 630 ' ... or computer?
560 R$=" ":T$="Press <SPACE> to compare the two hands...":GOSUB 5030
570 R=5 : FOR I=1 TO 5 : D=C(1,I):C=I*11+15:GOSUB 5340 : NEXT ' turn up cards
580 WP=1: T$=CVH$ : GOSUB 1370  ' show value of computer's hand
590 IF CVH=PVH THEN PB=PB+POT/2:CB=CB+POT/2 : T$="A draw!  Nobody wins."
600 IF CVH<PVH THEN PB=PB+POT : T$="Player wins"
610 IF CVH>PVH THEN CB=CB+POT : T$="Computer wins"
620 GOSUB 5120 : POT=0
630 R$=" ":T$="Press <SPACE> to pay off bets":GOSUB 5030: GOSUB 1420
640 FOR I=0 TO 6
650 LOCATE 5+I,26 : PRINT SPC(55); : LOCATE 14+I,26 : PRINT SPC(55);
660 NEXT : T$="" : WP=1 : GOSUB 1370 : WP=2 : GOSUB 1370
670 BROKE=0 : IF PB>A AND CB>A THEN 710  ' enough money to continue?
680 IF CB<=A THEN T$="Computer is busted!"
690 IF PB<=A THEN T$="You are busted!  Sorry..."
700 GOSUB 5120 : BROKE=-1 : RETURN
710 T$="":GOSUB 5120:R$=" ":T$="Press <SPACE> for next hand":GOSUB 5030
720 RETURN
730 '
740 ' process bets
750 '
760 F=0 : GOTO 830 ' computer first
770 F=1 ' player first
780 R$="":T$="Enter your bet or just press Enter to check: ":GOSUB 5030
790 B=VAL(A$) : IF A$="" OR B=0 THEN 820 ' check?
800 IF B<1 OR B>PB THEN BEEP : GOTO 780  ' valid bet?
810 PB=PB-B:PBET=B:POT=POT+B:GOSUB 1420:GOTO 920
820 IF F<1 THEN T$="Player checks." : GOSUB 5120 : RETURN
830 T$="Computer's turn to bet" : GOSUB 5120 : GOSUB 1520
840 B=CP:CB=CB-B:CBET=B:POT=POT+B : IF B>0 THEN GOSUB 1420:GOTO 860
850 T$="Computer checks" : GOSUB 5120 : IF F>0 THEN RETURN ELSE 780
860 T$="Computer bets $"+STR$(B) : GOSUB 5120
870 R$="":T$="Its $"+STR$(B)+" to you.  Call, raise, or press Enter to fold: "
880 GOSUB 5030 : PBET=VAL(A$) : IF PBET>0 AND PBET<B THEN BEEP:GOTO 870
890 PB=PB-PBET : POT=POT+PBET : B=PBET-B : GOSUB 1420
```

```
900 IF B=0 THEN T$="Player calls." : GOSUB 5120 : RETURN
910 IF B<0 THEN T$="Player folds!" : GOSUB 5120 : PBET=-1 : RETURN
920 T$="Its $"+STR$(B)+" to the computer." : GOSUB 5120 : GOSUB 1520
930 CBET=CP:CB=CB-CBET:POT=POT+CBET:B=CBET-B:GOSUB 1420
940 IF B=0 THEN T$="Computer calls." : GOSUB 5120 : RETURN
950 IF B<0 THEN T$="Computer folds." : GOSUB 5120 : CBET=-1 : RETURN
960 T$="Computer sees your $"+STR$(CBET-B)+" and raises you $"+STR$(B)
970 GOSUB 5120 : GOTO 870
980 '
990 ' player draws 1 to 3 cards
1000 '
1010 T$="You may draw one, two or three cards." : GOSUB 5120
1020 R$="":T$="Enter the digits 1 - 5 to denote which ones: ":GOSUB 5030
1030 IF A$="" THEN RETURN ELSE IF LEN(A$)>3 THEN BEEP : GOTO 1020
1040 FOR I=1 TO LEN(A$)
1050 IF INSTR("12345",MID$(A$,I,1))=0 THEN BEEP:GOTO 1010 ' any dupes?
1060 NEXT : R=14
1070 FOR I=1 TO LEN(A$)
1080 N=VAL(MID$(A$,I,1)):C=N*11+15:GOSUB 5470 ' cards down
1090 NEXT : GOSUB 1480
1100 FOR I=1 TO LEN(A$)
1110 N=VAL(MID$(A$,I,1)):P=P+1:D=D(P):C(2,N)=D:C=N*11+15:GOSUB 5340 ' cards up
1120 NEXT
1130 WP=2 :GOSUB 1620 : PVH=VH : T$=CV$ : GOSUB 1370 ' new hand value
1140 T$="":GOSUB 5120 : RETURN
1150 '
1160 ' deal and sort two hands of 5 cards: C(1,i)=computer, C(2,i)=player
1170 '
1180 POT=2*A : PB=PB-A : CB=CB-A : GOSUB 1420
1190 FOR I=1 TO 5 : P=P+1 : C(1,I)=D(P) : P=P+1 : C(2,I)=D(P) : NEXT ' deal'em
1200 FOR PL=1 TO 2 : FOR I=1 TO 4 : FOR J=I TO 5
1210 IF (C(PL,I) MOD 100)<(C(PL,J) MOD 100) THEN SWAP C(PL,I),C(PL,J) ' sort'em
1220 NEXT : NEXT : NEXT
1230 FOR I=1 TO 5
1240 D=C(1,I) : R=5: C=I*11+15 : GOSUB 5470 ' show 'em ... computer's down.
1250 D=C(2,I) : R=14:C=I*11+15 : GOSUB 5340 ' and the player's up
1260 NEXT
1270 WP=2:GOSUB 1620:PVH=VH:T$=CV$:GOSUB 1370 ' value player's hand
1280 WP=1:GOSUB 1620:CVH=VH:CDC$=DC$  ' and computer's
1290 IF CVH<210 AND SC=4 THEN CDC$=SCD$ : RETURN
1300 IF CVH<210 AND FC=4 THEN CDC$=FCD$ : RETURN
1310 IF VCH>210 AND LEN(CDC$)>2 AND RND>.5 THEN CDC$=LEFT$(CDC$,4) : RETURN
1320 IF (VCH\300)=3 AND RND>.8 THEN CDC$="" : RETURN
1330 RETURN
1340 '
1350 ' display value of hand
1360 '
1370 R=11:C=1 : IF WP=2 THEN R=20
1380 LOCATE R,C:PRINT SPC(30); : LOCATE R,C : PRINT T$ : RETURN
1390 '
1400 ' display bets, etc.
1410 '
1420 LOCATE 3,10:COLOR 0,7:PRINT USING " ######"; POT : COLOR 7,0
1430 LOCATE 8,5:PRINT USING "######"; CB : LOCATE 17,5:PRINT USING "######":PB
1440 RETURN
1450 '
1460 ' pause
1470 '
1480 FOR I=1 TO 2000 : NEXT I : RETURN
```

```
1490 '
1500 ' computer ponders player's bet
1510 '
1520 GOSUB 1480 : CP=CB-B : IF CP<0 THEN CP=0:RETURN ELSE IF B>0 THEN 1560
1530 IF CVH>400 THEN CP=INT((.2+RND*.3)*CB) : RETURN
1540 IF CVH>210 THEN CP=INT((.08*RND*.2)*CB) : RETURN
1550 IF RND>.5  THEN CP=INT((.05+RND*.15)*CB) : RETURN ELSE CP=0 : RETURN
1560 IF CVH>400 THEN CP=B+INT((.2+RND*.3)*CP) : RETURN
1570 IF CVH>210 THEN CP=B+INT((.08+RND*.2)*CP) : RETURN
1580 IF RND>.5  THEN CP=B : RETURN ELSE CP=0 : RETURN
1590 '
1600 ' get value of hand, and determine computer's discards
1610 '
1620 FOR I=1 TO 5
1630 CSP(I)=I : CVP(I)=I : C=C(WP,I) : CS(I)=C\100 : CV(I)=C MOD 100
1640 NEXT
1650 FOR I=1 TO 4 : FOR J=I TO 5
1660 IF CS(I)<CS(J) THEN SWAP CS(I),CS(J) : SWAP CSP(I),CSP(J) ' sort suits,
1670 IF CV(I)<CV(J) THEN SWAP CV(I),CV(J) : SWAP CVP(I),CVP(J) ' and values
1680 NEXT : NEXT
1690 C1=CV(1) : C2=CV(2) : C3=CV(3) : C4=CV(4) : C5=CV(5)
1700 CV12=C1=C2 : CV23=C2=C3 : CV34=C3=C4 : CV45=C4=C5
1710 CS12=CS(1)=CS(2) : CS23=CS(2)=CS(3) : CS34=CS(3)=CS(4) : CS45=CS(4)=CS(5)
1720 FC=0 : SC=0 : L4=0 : L3=0 : L32=0 : L22=0 : L2=0
1730 ' check 4 or 5 card flushes
1740 IF CS12 AND CS23 AND CS34 THEN FC=4 : FCD$=STR$(CSP(5)) : HC=1
1750 IF CS23 AND CS34 AND CS45 THEN FC=4 : FCD$=STR$(CSP(1)) : HC=2
1760 IF CS12 AND CS23 AND CS34 AND CS45 THEN FC=5 : FCD$="" : HC=1
1770 IF FC THEN HC=C(WP,CSP(HC)) : GOSUB 2230 : FHC$=HC$ : FHC=HC
1780 ' check 4 or 5 card straights
1790 IF C1-C2=1 AND C2-C3=1 AND C3-C4=1 THEN SC=4 : SCD$=STR$(CVP(5)) : HC=1
1800 IF C2-C3=1 AND C3-C4=1 AND C4-C5=1 THEN SC=4 : SCD$=STR$(CVP(1)) : HC=2
1810 IF C1-C2=1 AND C2-C3=1 AND C3-C4=1 AND C4-C5=1 THEN SC=5:SCD$="" : HC=1
1820 IF SC THEN HC=C(WP,CVP(HC)) : GOSUB 2230 : SHC$=HC$ : SHC=HC
1830 ' check for "like" cards: four of a kind?
1840 IF CV12 AND CV23 AND CV34 THEN L4=1 : LD4$=STR$(CVP(5)) : HC=1
1850 IF CV23 AND CV34 AND CV45 THEN L4=1 : LD4$=STR$(CVP(1)) : HC=2
1860 IF L4 THEN HC=C(WP,CVP(HC)) : GOSUB 2230 : HC4$=HC$ : HC4=HC
1870 ' ... full house?
1880 IF (CV12 AND CV34 AND CV45) THEN L32=1 : HC=5
1890 IF (CV12 AND CV23 AND CV45) THEN L32=1 : HC=1
1900 IF L32 THEN HC=C(WP,CVP(HC)) : GOSUB 2230 : HC32$=HC$ : HC32=HC
1910 ' ... 3 of a kind?
1920 IF CV12 AND CV23 THEN L3=1:LD3$=STR$(CVP(4))+STR$(CVP(5)):HC=1
1930 IF CV34 AND CV45 THEN L3=1:LD3$=STR$(CVP(1))+STR$(CVP(2)):HC=3
1940 IF CV23 AND CV34 THEN L3=1:LD3$=STR$(CVP(1))+STR$(CVP(5)):HC=2
1950 IF L3 THEN HC=C(WP,CVP(HC)) : GOSUB 2230 : HC3$=HC$ : HC3=HC
1960 ' ... 2 pair?
1970 IF CV12 AND CV34 THEN L22=1 : LD22$=STR$(CVP(5)) : HC22=1:HC21=3
1980 IF CV12 AND CV45 THEN L22=1 : LD22$=STR$(CVP(3)) : HC22=1:HC21=4
1990 IF CV23 AND CV45 THEN L22=1 : LD22$=STR$(CVP(1)) : HC22=2:HC21=4
2000 IF L22=0 THEN 2040
2010 HC=C(WP,CVP(HC21)):GOSUB 2230:HC22$=HC$
2020 HC=C(WP,CVP(HC22)):GOSUB 2230:HC22$=HC$+"s over"+HC22$+"s":HC22=HC
2030 ' ... 1 pair?
2040 IF CV12 THEN L2=1:LD2$=STR$(CVP(3))+STR$(CVP(4))+STR$(CVP(5)):HC=1
2050 IF CV23 THEN L2=1:LD2$=STR$(CVP(1))+STR$(CVP(4))+STR$(CVP(5)):HC=2
2060 IF CV34 THEN L2=1:LD2$=STR$(CVP(1))+STR$(CVP(2))+STR$(CVP(5)):HC=3
2070 IF CV45 THEN L2=1:LD2$=STR$(CVP(1))+STR$(CVP(2))+STR$(CVP(3)):HC=4
```

```
2080 IF L2 THEN HC=C(WP,CVP(HC)) : GOSUB 2230 : HC2$=HC$ : HC2=HC
2090 ' ... high card!
2100 LD1$=STR$(CVP(3))+STR$(CVP(4))+STR$(CVP(5))
2110 HC=C(WP,CVP(1)) : GOSUB 2230 : HC1$=HC$ : HC1=HC
2120 ' set value of hand, discards and description: VH, CV$, DC$
2130 IF FC=5 AND SC=5 THEN VH=900+FHC:CV$=FHC$+" high straight flush" : RETURN
2140 IF L4=1  THEN VH=800+HC4 :DC$=LD4$:CV$="Four"+HC4$+"s":RETURN
2150 IF L32=1 THEN VH=700+HC32:CV$=HC32$+" high full house" : RETURN
2160 IF FC=5  THEN VH=600+FHC :CV$=FHC$+" high flush" : RETURN
2170 IF SC=5  THEN VH=500+SHC :CV$=SHC$+" high straight" : RETURN
2180 IF L3=1  THEN VH=400+HC3 :DC$=LD3$ : CV$="Three"+HC3$+"s":RETURN
2190 IF L22=1 THEN VH=300+HC22:DC$=LD22$ : CV$="2 pair,"+HC22$ : RETURN
2200 IF L2=1  THEN VH=200+HC2 :DC$=LD2$ : CV$="Pair of"+HC2$+"s":RETURN
2210 VH=100+HC1 : DC$=LD1$ : CV$=HC1$+" high" : RETURN
2220 ' card name
2230 HC=HC MOD 100 : IF HC=1 THEN HC=14
2240 IF HC<=10 THEN HC$=STR$(HC) : RETURN
2250 HC$=" "+MID$("Jack QueenKing Ace",HC*5-54,5+(HC=11 OR HC=13)): RETURN
2260 '
2270 ' computer's draw
2280 '
2290 IF LEN(CDC$)=0 THEN T$="stay pat." ELSE D=LEN(CDC$)\2:T$="take"+STR$(D)
2300 T$="Computer's turn to draw.  It decides to "+T$ : GOSUB 5120
2310 FOR I=1 TO D : P=P+1 : C(1,VAL(MID$(CDC$,I*2,1)))=D(P) : NEXT
2320 WP=1 : GOSUB 1620 : CVH=VH : CVH$=CV$ : RETURN ' new hand value
5000 '
5010 ' process player responses
5020 '
5030 LOCATE 23,40-LEN(T$)/2 : PRINT T$;
5040 IF R$="" THEN LINE INPUT;"",A$ : GOTO 5080 ' multi-char reply
5050 C$="" : WHILE C$="" : C$=INKEY$ : WEND  ' or single character
5060 IF C$>"Z" THEN C$=CHR$(ASC(C$)-32)      ' shift to upper case
5070 IF INSTR(R$,C$)=0 THEN BEEP : GOTO 5050 ' validate character
5080 LOCATE 23,1 : PRINT SPC(79); : RETURN
5090 '
5100 ' display a message
5110 '
5120 LOCATE 21,1 : PRINT SPC(79);
5130 LOCATE 21,40-LEN(T$)/2 : PRINT T$; : RETURN
5140 '
5150 ' set up deck
5160 '
5170 NC=ND*52-1 : DIM D(NC) : P=-1 ' number cards, deck array and pointer
5180 FOR I=1 TO ND : FOR J=1 TO 4  ' for each deck and suit
5190   FOR K=2 TO 14 : P=P+1 : D(P)=J*100+K : NEXT ' set card numbers
5200 NEXT : NEXT
5210 P=ND*52 : RETURN ' set pointer to force shuffle
5220 '
5230 ' shuffle the cards
5240 '
5250 T$="Excuse me while I shuffle the deck..." : GOSUB 5120
5260 FOR I=1 TO 3 : FOR P=0 TO NC  ' three passes over deck
5270 SWAP D(P),D(INT(RND*NC))    ' exchange each card with a random one
5280 NEXT : NEXT : P=-1          ' set deck pointer when done
5290 T$="":GOSUB 5120:R$=" ":T$="All done. Press <SPACE> to play":GOSUB 5030
5300 RETURN
5310 '
5320 '   Face of 6x8 card
```

```
5330 '
5340 COLOR ABS((D\100)<3)*4,7
5350 F1$=MID$("*2345678910JQKA",D MOD 100,1):F2$=F1$
5360 IF (D MOD 100)<>10 THEN F1$=F1$+" ":F2$=" "+F2$ ELSE F1$="10":F2$="10"
5370 LOCATE R+0,C:PRINT CHR$(218);STRING$(6,196);CHR$(191);
5380 LOCATE R+1,C:PRINT CHR$(179);F1$+"    ";CHR$(179);
5390 LOCATE R+2,C:PRINT CHR$(179);" ";CHR$(D\100+2);"    ";CHR$(179);
5400 LOCATE R+3,C:PRINT CHR$(179);"    ";CHR$(D\100+2);" ";CHR$(179);
5410 LOCATE R+4,C:PRINT CHR$(179);"    "+F2$;CHR$(179);
5420 LOCATE R+5,C:PRINT CHR$(192);STRING$(6,196);CHR$(217);
5430 COLOR 7,0:RETURN
5440 '
5450 '   Back of 6x8 card
5460 '
5470 COLOR 0,7:LOCATE R,C:PRINT CHR$(201);STRING$(6,205);CHR$(187)
5480 FOR K=R+1 TO R+4
5490 LOCATE K,C:PRINT CHR$(186);STRING$(6,176);CHR$(186)
5500 NEXT
5510 LOCATE R+5,C:PRINT CHR$(200);STRING$(6,205);CHR$(188);
5520 COLOR 7,0:RETURN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

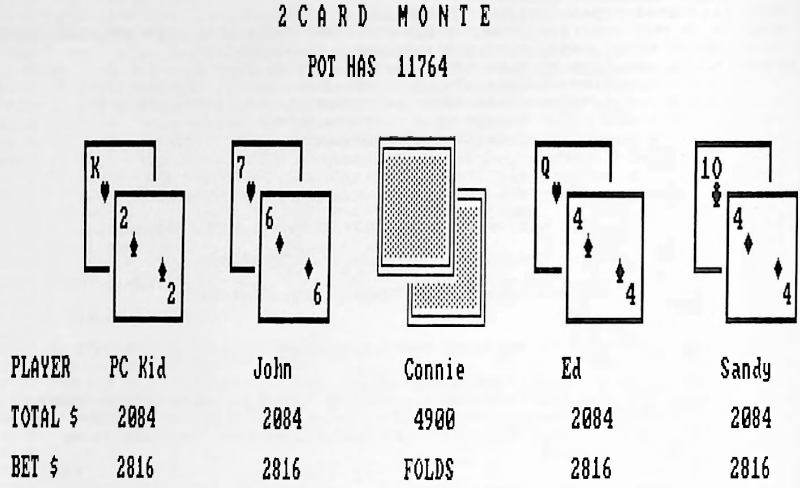
Program 5.2 DRAW - 5 card draw poker

Two-Card Monte

This program is a multiplayer poker game. The computer, also known as the PC Kid, and the other players (up to four) are each dealt two cards, one up and one down. After the deal, each player takes a turn looking at his down card; then everyone bets in sequence. You may either bet, raise, or fold. The manner of betting is identical to the procedure used in the five-card draw poker game. The computer keeps track of who raises and what bet each player must make to stay in the hand.

When all bets are down, there is a showdown. The high hand wins—perhaps a pair of queens or 8's. If there are no pairs, the hand with the highest card wins. The payoff is made to the winning player, and the game continues. With each new hand, the betting begins with the next player in the game so everybody takes turns at betting first. See Figure 5.4 for an example of the game being played.

FIGURE 5.4. Two-card monte multiplayer poker game.


2 CARD MONTE
POT HAS 11764
PLAYER PC Kid John Connie Ed Sandy
TOTAL $ 2084 2084 4900 2084 2084
BET $ 2816 2816 FOLDS 2816 2816
PC Kid, you won!!!
Press <SPACE> to pay off bets...

In the showdown, it is common that a jack or queen high can win, so do not play in hopes of getting a match to make a pair. On the other hand, any pair is a good hand, so bet strong if you get one—but beware! The PC Kid is a notorious bluffer! There is also the possibility of two or more winning hands, or in other words, there is a tie. In this case, the pot is split among the winners.

The computer shuffles the cards only after the deck is exhausted, not after every hand. Thus, part of the playing strategy is to remember cards already played. Multiple decks can also be used in the game. One of the initial program startup questions asks for the number of decks to be used. Enter a number from 1 to 4. This keeps the number of shuffles down and also increases the variety of hands possible.

This game illustrates the intricacies of programming a multiplayer card game. It requires a number of arrays that keep track of

all values associated with each player in the game: BR for the bankrolls, BP has the player's bets, N$ contains the players' names, VH has the value of the hands, and C contains the two cards in each hand.

The BR array is used when players are asked if they want to bet. When someone goes bankrupt, BR will be 0. To determine who bet during a hand, the program must check the BP array. This is a necessary step because players can fold from a hand but still have money to play the next hand. The player name array, N$, allows the computer to ask you by name if you want to bet. Also, when the winning hand or hands is determined, it announces by name the player who has won.

The last array, C, is two dimensional: The first subscript is the player index and the second subscript is a 1 and 2 for the two cards in the player's hand. This is similar to the C array used in the draw game but modified for five players, each with two cards.

Hands are evaluated by assigning a value equal to the high card in the hand times 100 plus the second card. In the event the hand is a pair (if the cards are alike), then 1500 is added to one of the card's values. Thus, a 906 is a hand containing a 9 and a 6. A 1507 is a pair of 7's. When the hand comparison is made, the highest hand value is compared to all hand values to check that two or more players do not have the same hands.

Betting by the computer, or PC Kid, is similar to the previous card games. In this game, though, the computer can "see" the face-up cards of the other players. The highest face-up card among the players is found and taken into account as the computer determines a bet. The game also has some random number based decisions thrown in to simulate bluffing. The listing for the game appears in Program 5.3.

```
0 ' 2CARD  2 Card Monte  (Poker with 2 cards)
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 ' One to four players, and your PC, aka the "PC Kid", bet on poker hands
50 ' of 2 cards.  A pair of Aces is tops, but a Queen or King high is good.
60 ' Everyone may check, bet, raise or fold.
70 '
80 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
90 DEFINT A-Z : DEFSNG B : RANDOMIZE VAL(LEFT$(TIME$,2)+MID$(TIME$,4,2))
100 DIM BR(4), BP(4), N$(4), C(4,2), VH(4)
110 PRINT"2 Card Monte - Poker Game for 1 to 4 players" : PRINT
120 '
130 '  Get number of decks, players, their names and bankroll
```

```
140 '
150 PRINT "How many decks of cards (1-8)? ";
160 ND=0 : WHILE ND<1 OR ND>8 : ND=VAL(INKEY$) : WEND
170 PRINT ND : GOSUB 5170 : PRINT ' set-up decks
180 PRINT "How many players (1-4)? ";
190 NP=0 : WHILE NP<1 OR NP>4 : NP=VAL(INKEY$) : WEND
200 PRINT NP : PRINT
210 FOR PL=1 TO NP
220   PRINT "Enter player";PL;"'s name: "; : LINE INPUT N$
230   IF N$="" THEN BEEP : GOTO 220
240   IF ASC(N$)>96 THEN N$=CHR$(ASC(N$)-32)+MID$(N$,2)
250   N$(PL)=LEFT$(N$,8)
260 NEXT : N$(0)="PC Kid"
270 LINE INPUT"Enter the player's bankroll amount: ",N$
280 B=VAL(N$) : IF B<1 OR B>100000! THEN BEEP : GOTO 270
290 FOR PL=0 TO NP : BR(PL)=B:BP(PL)=0 : NEXT
300 LINE INPUT"Enter the ante per hand: ",N$
310 A=VAL(N$) : IF A<10 OR A>B/10 THEN BEEP : GOTO 300
320 GOSUB 5250 : CLS ' initial shuffle
330 '
340 ' set-up initial screen
350 '
360 PRINT TAB(28);"2 C A R D   M O N T E" : POT=0 : GOSUB 1460
370 LOCATE 15,1:PRINT"PLAYER":PRINT:PRINT"TOTAL $":PRINT:PRINT"BET $"
380 FOR PL=0 TO NP
390 GOSUB 1480 : GOSUB 1490 ' display player names and bankrolls
400 NEXT PL
410 FP=-1 ' init for first player to bet
420 '
430 ' deal'em
440 '
450 IF P>=(ND*52-NP*2-2) THEN GOSUB 5250 ' time to shuffle?
460 FOR PL=0 TO NP
470 IF BR(PL)<A THEN 510 ' gotta have enough to ante
480 P=P+1 : D=D(P) : C(PL,1)=D : R=6 : C=PL*15+8  : GOSUB 5340 ' up card
490 P=P+1 : D=D(P) : C(PL,2)=D : R=8 : C=PL*15+11 : GOSUB 5470 ' down
500 POT=POT+A : BR(PL)=BR(PL)-A : GOSUB 1490 ' show ante
510 NEXT : GOSUB 1460  ' show pot
520 '
530 ' display each player's down card
540 '
550 FOR PL=1 TO NP
560 IF BP(PL)<0 THEN 620 ' bankrupt?
570 COLOR 0,7 : GOSUB 1480 : COLOR 7,0
580 R$=" ":T$=N$(PL)+", press <SPACE> to see your down card":GOSUB 5030
590 D=C(PL,2) : C=PL*15+11 : GOSUB 5340
600 T$="Now press <SPACE> to turn it back over":GOSUB 5030
610 GOSUB 5470 : GOSUB 1480
620 NEXT
630 '
640 ' computer checks its hand, and player up cards
650 '
660 IF BP(0)<0 THEN 740  ' PC still in the game?
670 MPC=0 : C=0 : PL=0 : GOSUB 1690 : VCH=VH  ' value of computer's hand
680 FOR I=1 TO NP : IF BP(I)<0 THEN 700       ' bypass players not betting
690 CV=C(I,1) MOD 100 : IF MPC<CV THEN MPC=CV ' maximum player up card
700 NEXT : MPC=MPC*100
```

```
710 '
720 ' get the bets, checks, raises or folds
730 '
740 PL=FP : BET=0
750 FOR I=0 TO NP : PL=(PL+1) MOD (NP+1)
760 IF BP(PL)<0 THEN 910 ' bankrupt?
770 COLOR 0,7 : GOSUB 1480 : COLOR 7,0
780 IF BET=0 THEN T$=", enter bet or just press Enter to check: " : GOTO 810
790 IF BET=BP(PL) THEN T$=", either raise or press Enter to check: " : GOTO 810
800 IF BET>BP(PL) THEN T$=", bet $"+STR$(BET-BP(PL))+" or more, or press Enter t
o fold: "
810 T$=N$(PL)+T$ : R$=""
820 IF PL>0 THEN GOSUB 5030 : B=VAL(A$)  ' process player bets, or
830 IF PL=0 THEN GOSUB 5120 : GOSUB 1810 : GOSUB 1770  ' the PC bets
840 IF B<0 OR B>BR(PL) OR (B>0 AND B+BP(PL)<BET) THEN BEEP:GOTO 810
850 IF B=0 AND BET=0 THEN GOSUB 1650 : GOTO 900  ' check no bets made
860 IF B=0 AND BET>0 THEN GOSUB 1590 : GOTO 900  ' folds
870 IF B+BP(PL)>BET THEN BET=BP(PL)+B:I=0  ' raise, let everybody bet again
880 POT=POT+B : GOSUB 1460
890 BP(PL)=BP(PL)+B : GOSUB 1500 : BR(PL)=BR(PL)-B : GOSUB 1490
900 GOSUB 1480 : IF PL=0 THEN T$="" : GOSUB 5120
910 NEXT
920 FP=(FP+1) MOD (NP+1) ' next player who bets first
930 '
940 ' bets down, check for folds, and who won
950 '
960 FOLDS=0 : FOR PL=0 TO NP : FOLDS=FOLDS-(BP(PL)<0) : NEXT ' who folded
970 IF FOLDS<NP THEN 1020
980 T$="All but one player folded!" : GOSUB 5120 : GOSUB 1770
990 FOR PL=0 TO NP
1000 IF BP(PL)<>-1 THEN T$=N$(PL)+"*****":TIE=1:VH(PL)=1:WH=1:GOTO 1190
1010 NEXT
1020 R$=" ":T$="Pot's right.  Press <SPACE> for showdown" : GOSUB 5030
1030 '
1040 ' which hand or hands won?
1050 '
1060 WH=0 : WP=0 : TIE=0 : R=8
1070 FOR PL=0 TO NP
1080 IF BP(PL)<0 THEN 1120
1090 D=C(PL,2) : C=PL*15+11 : GOSUB 5340 ' display down cards
1100 GOSUB 1690 : VH(PL)=VH       ' get value of hand for player PL
1110 IF VH>WH THEN WH=VH : WP=PL ' highest value is winning hand
1120 NEXT : T$=""
1130 '
1140 ' display winning hands, and who...
1150 '
1160 FOR PL=0 TO NP
1170 IF VH(PL)=WH THEN TIE=TIE+1 : T$=T$+N$(PL)+" and " ' may be a tie
1180 NEXT
1190 IF TIE=1 THEN T$=LEFT$(T$,LEN(T$)-5)+", you won!!!"
1200 IF TIE>1 THEN T$=LEFT$(T$,LEN(T$)-5)+" win and split pot!"
1210 GOSUB 5120:R$=" ":T$="Press <SPACE> to pay off bets...":GOSUB 5030
1220 '
1230 ' pay off winner(s) and count who's broke
1240 '
1250 POT=POT\TIE : BROKE=0
1260 FOR PL=0 TO NP
```

```
1270 IF VH(PL)=WH THEN BR(PL)=BR(PL)+POT ' player won
1280 VH(PL)=0 : BP(PL)=0 : GOSUB 1490 : GOSUB 1500
1290 IF BR(PL)<A THEN GOSUB 1540  ' player busted (can't ante)
1300 NEXT
1310 '
1320 ' check if end of game otherwise play again
1330 '
1340 POT=0 : GOSUB 1460 : T$="" : GOSUB 5120
1350 IF BROKE=NP THEN T$="All but one player is busted!":GOSUB 5120 : GOTO 1390
1360 R$=" ":T$="Press <SPACE> for next hand" : GOSUB 5030
1370 FOR R=6 TO 13 : LOCATE R,1 : PRINT SPC(79); : NEXT
1380 GOTO 450
1390 R$="YN":T$="Want to play some more (Y or N)? ":GOSUB 5030
1400 IF C$="Y" THEN CLS : GOTO 270  ' play some more
1410 IF C$="N" THEN C$=CHR$(27) : GOTO 9050 ' exit from game
1420 BEEP : GOTO 1390
1430 '
1440 ' subroutines to display money, and player names
1450 '
1460 COLOR 0,7 : LOCATE 3,28 : PRINT USING"   POT HAS  #####     ";POT
1470 COLOR 7,0 : RETURN
1480 LOCATE 15,PL*15+10 : PRINT USING" \     \ ";N$(PL) : RETURN
1490 LOCATE 17,PL*15+10 : PRINT USING"######";BR(PL) : RETURN
1500 LOCATE 19,PL*15+10 : PRINT USING" #####";BP(PL) : RETURN
1510 '
1520 ' player blew it!
1530 '
1540 LOCATE 19,PL*15+10 : PRINT"BANKRUPT" ' message
1550 BROKE=BROKE+1 : BP(PL)=-1 : RETURN   ' set flags
1560 '
1570 ' player folds
1580 '
1590 LOCATE 19,PL*15+10 : PRINT " FOLDS " ' print message,
1600 BP(PL)=-1:R=6:C=PL*15+8:GOSUB 5470   ' set flag, and turn over up-card
1610 RETURN
1620 '
1630 ' player checks with no bet
1640 '
1650 LOCATE 19,PL*15+10:PRINT "CHECKS"; : RETURN
1660 '
1670 ' set value of hand, VH, for player PL
1680 '
1690 C1=C(PL,1) MOD 100 : C2=C(PL,2) MOD 100 ' get card values
1700 IF C1<C2 THEN SWAP C1,C2 ' put in order
1710 VH=C1*100+C2             ' value of hand
1720 IF C1=C2 THEN VH=C1+1500 ' check for pairs
1730 RETURN
1740 '
1750 ' momentary pause
1760 '
1770 FOR K=1 TO 2500 : NEXT : RETURN
1780 '
1790 ' PC ponders a bet; VCH is value of computer's hand
1800 '
1810 B=BR(0)-BET:IF B<0 THEN B=0:RETURN ELSE IF BET>0 THEN 1850
1820 IF VCH>1500 THEN B=INT((.2+RND*.3)*BR(0)) : RETURN ' pair?
1830 IF VCH>706 OR VCH>MPC THEN B=INT((.08+RND*.2)*BR(0)) : RETURN
1840 IF RND>.9 THEN B=INT((.05+RND*.15)*BR(0)) : RETURN ELSE B=0 : RETURN
1850 IF VCH<MPC AND RND<.95 THEN B=0 : RETURN
1860 IF VCH>1500 THEN B=BET+INT((.4+RND*.2)*B) : RETURN
1870 IF VCH>600 THEN B=BET : RETURN
1880 B=0 : RETURN
5000 '
```

```
5010 ' process player responses
5020 '
5030 LOCATE 23,40-LEN(T$)/2 : PRINT T$;
5040 IF R$="" THEN LINE INPUT;"",A$ : GOTO 5080 ' multi-char reply
5050 C$="" : WHILE C$="" : C$=INKEY$ : WEND  ' or single character
5060 IF C$>"Z" THEN C$=CHR$(ASC(C$)-32)      ' shift to upper case
5070 IF INSTR(R$,C$)=0 THEN BEEP : GOTO 5050 ' validate character
5080 LOCATE 23,1 : PRINT SPC(79); : RETURN
5090 '
5100 ' display a message
5110 '
5120 LOCATE 21,1 : PRINT SPC(79);
5130 LOCATE 21,40-LEN(T$)/2 : PRINT T$; : RETURN
5140 '
5150 ' set up deck
5160 '
5170 NC=ND*52-1 : DIM D(NC) : P=-1 ' number cards, deck array and pointer
5180 FOR I=1 TO ND : FOR J=1 TO 4  ' for each deck and suit
5190   FOR K=2 TO 14 : P=P+1 : D(P)=J*100+K : NEXT ' set card numbers
5200 NEXT : NEXT
5210 P=ND*52 : RETURN ' set pointer to force shuffle
5220 '
5230 ' shuffle the cards
5240 '
5250 T$="Excuse me while I shuffle the deck..." : GOSUB 5120
5260 FOR I=1 TO 3 : FOR P=0 TO NC  ' three passes over deck
5270 SWAP D(P),D(INT(RND*NC))      ' exchange each card with a random one
5280 NEXT : NEXT : P=-1            ' set deck pointer when done
5290 T$="":GOSUB 5120:R$=" ":T$="All done. Press <SPACE> to play":GOSUB 5030
5300 RETURN
5310 '
5320 '    Face of 6x8 card
5330 '
5340 COLOR ABS((D\100)<3)*4,7
5350 F1$=MID$("*23456789JQKA",D MOD 100,1):F2$=F1$
5360 IF (D MOD 100)<>10 THEN F1$=F1$+" ":F2$=" "+F2$ ELSE F1$="10":F2$="10"
5370 LOCATE R+0,C:PRINT CHR$(218);STRING$(6,196);CHR$(191);
5380 LOCATE R+1,C:PRINT CHR$(179);F1$+"     ";CHR$(179);
5390 LOCATE R+2,C:PRINT CHR$(179);" ";CHR$(D\100+2);"    ";CHR$(179);
5400 LOCATE R+3,C:PRINT CHR$(179);"     ";CHR$(D\100+2);" ";CHR$(179);
5410 LOCATE R+4,C:PRINT CHR$(179);"    "+F2$;CHR$(179);
5420 LOCATE R+5,C:PRINT CHR$(192);STRING$(6,196);CHR$(217);
5430 COLOR 7,0:RETURN
5440 '
5450 '    Back of 6x8 card
5460 '
5470 COLOR 0,7:LOCATE R,C:PRINT CHR$(201);STRING$(6,205);CHR$(187)
5480 FOR K=R+1 TO R+4
5490 LOCATE K,C:PRINT CHR$(186);STRING$(6,176);CHR$(186)
5500 NEXT
5510 LOCATE R+5,C:PRINT CHR$(200);STRING$(6,205);CHR$(188);
5520 COLOR 7,0:RETURN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 5.3 2CARD - 2 card poker game

Blackjack

The game of Blackjack has the distinction of being the only casino game in which the player can have an edge over the house. Also known as 21, the essence of the game is to get a card total as close to 21 as possible without going over 21. You play against a dealer (the "house"), who plays for a minimum card value of 17. The dealer stays (draws no more cards) on all values of 17 and higher.

Cards are counted based on their numeric face values with an ace being either 1 or 11, and all face cards being 10. This causes a slight change in our deck setup subroutine at line 5140. The card numbers range from 1 to 13 instead of 2 to 14, which changes the FOR / NEXT loop at line 5190. The program also checks for all card numbers that are 11 or greater and converts them to 10 when calculating the total value of a hand.

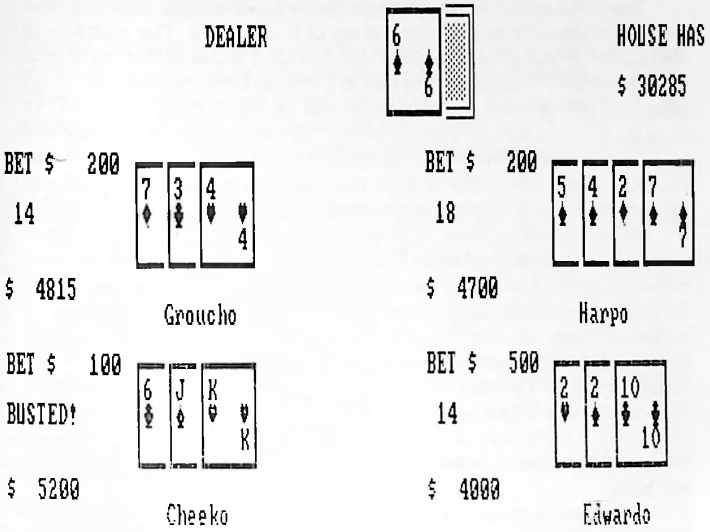
In our game, from one to four players may play against the dealer. See Figure 5.5 for an illustration of the game in progress. When the game begins, the program first asks how many decks of cards are to be used. Many casinos use as many as eight decks to compensate for the edge that some players can achieve through skillful play; others use two to six decks. (There are very few one-deck blackjack games left in casinos.)

The program asks for the number of players, their bankrolls, and their names. Then the game begins. Two cards are initially dealt face down to everyone, and each player makes his bets. All cards are turned face up, except for one of the dealer's cards (the "hole card"). Each player is told his hand's total, and is asked by name if he wants a hit or to stay. The player must press either an H for a hit (another card) or an S to stay (the player gets no more cards and stays with what he has at that point).

On the very first query to each player, an additional question of whether the player wants to "double down" is also asked. A player can press a D to double his initial bet—he'll then get only one more card for this hand. This is almost always recommended if a hand totals 11, and for some totals of 10 and less. Part of skillful play is knowing when to double down.

If a player draws cards and goes beyond 21, he is "busted" and loses his bet, regardless of what the dealer does—even if he, too, busts. Since an ace counts as 1 or 11, hands with one or more aces may have two values. This is known as a "soft total" since a 5 and an

FIGURE 5.5. Blackjack game example.



ace is either a 6 or a 16, depending on how you count the ace. Adding a 9 to this hand makes it a "hard total" of 15 since, if the ace were now an 11, the total would be over 21. Our program keeps track of all of this for you when it asks you to bet.

After all hands are played, the dealer draws cards until a total of 17 or more is achieved. If the dealer busts, then all players who did not bust win. If the dealer does not bust, players with hand totals greater than the dealer's hand win. A tie with the dealer is known as a push, and no money is won or lost. Note that there are two types of 21: a natural, a two-card ace and a facecard; and a multicard total of 21. If both player and dealer hands are 21, then the hand that is a natural wins.

This game uses many of the techniques developed in the previous multiplayer card game. The notable exception is that the computer, or dealer, does no betting. In particular, the same arrays are

used for the player bets, bankroll, names, and cards: BP, BR, N$, C.

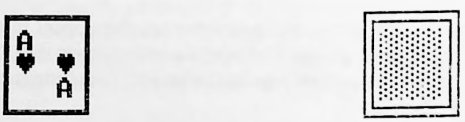
The card array is handled somewhat differently, though. The second subscript takes on the values of 0, 1, and 2. The number of cards dealt to this player is at C(PL,0); the total value of this hand is at C(PL,1); and C(PL,2) has a flag that is 0 or 1, indicating whether or not an ace is in this hand and if the ace will be counted as a 1, or an 11 (that is, if it is a soft or a hard hand). These three values allow the final totals of the players and the dealer to be compared. The number of cards is needed to check for a natural 21. The hard or soft flag is used to make sure the ace is counted properly.

When drawing cards, a player may potentially take up to six or more cards and still not bust. This means that since there may be up to four players, the overall screen layout must be carefully designed. To fit all of the cards on the screen requires that the cards overlap by a few columns so only the card value shows. Furthermore, our 6-rows × 8-columns card image used in the previous games is too large. So we will modify our card image subroutines to be 5 rows by 6 columns. See Figure 5.6 for an example of the new card format. Thus, the subroutines at lines 5340 and 5470 have been changed slightly. The other routines for messages, prompts, and shuffling are the same.

All player information is formatted to fit in four distinct areas on the screen. The program takes the player number in variable PL and calculates a row and column for the cards, the names of the players, each player's bet, and so on. Players 0 and 2 are at column 10, and players 1 and 3 are at column 50. The row positions are 7 for players 0 and 1, and 15 for players 2 and 3. The column position must also be offset by a value of 3 for each card displayed. This causes the cards in the hands to overlap slightly as they are displayed.

To calculate these positions requires using an integer divide, \, and the MOD operators with the player number in PL and the card number CN:

FIGURE 5.6. Image of a 5-row by 6-column card.



```
R = ( PL\2 ) * 8 + 7
C = ( PL MOD 2 ) * 40 + 10 + CN * 3
```

Similar calculations are used when displaying the bet amounts, the bankroll, and the card total. Player number 4 is the dealer. Dealer information is displayed at a fixed position at the top–center of the screen. The above calculations are not needed.

The complete blackjack game is listed in Program 5.4.

```
0 ' BJACK  Blackjack Card Game
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 ' Blackjack for 1 to 4 players.  Closest to 21 wins; Blackjack pays
50 ' 3 to 2. You may also double down on first card only. There is no
60 ' insurance, surrender, or splitting of pairs.
70 '
80 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
90 RANDOMIZE VAL(LEFT$(TIME$,2)+MID$(TIME$,4,2))
100 DEFINT A-Z : DEFSNG B  : DIM BR(3),BP(3),N$(3),C(4,2)
110 PRINT"Blackjack - Casino Card Game for 1 to 4 players" : PRINT
120 '
130 '  Get number of decks, players, their names and bankroll
140 '
150 PRINT "How many decks of cards (1-8): ";
160 ND=0 : WHILE ND<1 OR ND>8 : ND=VAL(INKEY$) : WEND
170 PRINT ND : GOSUB 5170 : PRINT ' set-up decks
180 PRINT "How many players (1-4)? ";
190 NP=0 : WHILE NP<1 OR NP>4 : NP=VAL(INKEY$) : WEND
200 PRINT NP : NP=NP-1 : PRINT
210 FOR PL=0 TO NP
220   PRINT "Enter player";PL+1;"'s name: "; : LINE INPUT N$
230   IF N$="" THEN BEEP : GOTO 220
240   IF N$>"Z" THEN N$=CHR$(ASC(N$)-32)+MID$(N$,2)
250   N$(PL)=LEFT$(N$,8)
260 NEXT
270 LINE INPUT"Enter player's bankroll: ",N$
280 B=VAL(N$) : IF B<1 OR B>100000! THEN BEEP : GOTO 270
290 FOR PL=0 TO NP : BR(PL)=B : NEXT
300 BANK!=2*B*(NP+1) ' house bankroll
310 GOSUB 5250 : CLS ' initial shuffle
320 '
330 ' Format screen
340 '
350 LOCATE 2,20 : PRINT"DEALER"; : LOCATE ,60 : PRINT"HOUSE HAS";
360 FOR PL=0 TO NP :
370 LOCATE (PL\2)*8+7,(PL MOD 2)*40+1 : PRINT "BET"
380 GOSUB 1280 : GOSUB  1260
390 NEXT
400 '
410 ' Display 2 down cards per player
420 '
430 FOR PL=0 TO NP : IF BR(PL)<=0 THEN 460
440 R=(PL\2)*8+7 : C=(PL MOD 2)*40+1
450 C=C+12 : GOSUB 5470 : C=C+3 : GOSUB 5470 ' player's cards
460 NEXT
470 GOSUB 1250 : R=1 : C=37 : GOSUB 5470 : C=40 : GOSUB 5470 ' dealer's
```

```
480 '
490 ' Get initial bets
500 '
510 FOR PL=0 TO NP : IF BR(PL)<=0 THEN 570
520 COLOR 0,7 : GOSUB 1280 : COLOR 7,0
530 R$="":T$=N$(PL)+", what is your bet? ":GOSUB 5030
540 IF A$="" OR LEN(A$)>5 THEN BEEP : GOTO 530
550 B=VAL(A$) : IF B<1 OR B>BR(PL) THEN BEEP : GOTO 530
560 BP(PL)=B : BR(PL)=BR(PL)-B : GOSUB 1260 : GOSUB 1270 : GOSUB 1280
570 NEXT
580 FOR PL=0 TO 4 : FOR I=0 TO 2 : C(PL,I)=0 : NEXT : NEXT
590 '
600 ' deal 1st two cards
610 '
620 FOR CN=1 TO 2   ' for cards 1 and 2
630 FOR PL=0 TO NP : IF BP(PL)=0 THEN 650 ' player broke?
640 GOSUB 1340 : GOSUB 1430  ' display card, save total
650 NEXT PL : NEXT CN
660 CN=1 : PL=4 : GOSUB 1340 : GOSUB 1430 ' dealer's card
670 '
680 ' players do their thing
690 '
700 FOR PL=0 TO NP : IF BP(PL)=0 THEN 820 ' player broke?
710 GOSUB 1480 : CT$=T$ : IF C(PL,1)=21 THEN 810 ' hand total, check for BJ
720 COLOR 0,7 : GOSUB 1280 : COLOR 7,0
730 R$="DHS":T$=N$(PL)+", you have "+CT$+". Double, Hit, or Stand (D,H,S)? ":GOS
UB 5030
740 IF C$="S" THEN 810
750 IF C$="D" AND BR(PL)<BP(PL) THEN BEEP:R$=" ":T$="You don't have enough money
 to double-down.  Press <SPACE> to continue": GOSUB 5030: GOTO 790
760 IF C$="D" THEN BR(PL)=BR(PL)-BP(PL):BP(PL)=2*BP(PL):GOSUB 1260:GOSUB 1270
770 CN=C(PL,0)+1 : GOSUB 1340 : GOSUB 1430 : GOSUB 1480 : CT$=T$
780 IF T>=21 OR C$="D" THEN 810
790 R$="HS":T$=N$(PL)+", you have "+CT$+". Hit, or Stand (H or S)? "
800 GOSUB 5030 : GOTO 740
810 GOSUB 1280
820 NEXT PL
830 '
840 ' dealer plays out its hand
850 '
860 PL=4 ; CN=C(PL,0) : T=0
870 WHILE T<=16
880   CN=CN+1 : GOSUB 1340 : GOSUB 1430 : GOSUB 1480
890   FOR I=1 TO 1000 : NEXT I ' pause
900 WEND
910 '
920 ' pay off bets, and see who's broke
930 '
940 R$=" ":T$="Press <SPACE> to pay off bets": GOSUB 5030
950 DT=C(4,1) : BROKE=0
960 FOR PL=0 TO NP : IF BP(PL)=0 THEN 1030
970 PT=C(PL,1)
980 IF PT>21 OR (PT<DT AND DT<=21) THEN T$="You lose ":
      BANK!=BANK!+BP(PL) : GOTO 1020
990 IF PT=21 AND C(PL,0)=2 AND NOT (DT=21 AND C(4,0)=2) THEN T$="You WIN!  ":
      BANK!=BANK!-INT(1.5*BP(PL)):BR(PL)=BR(PL)+INT(2.5*BP(PL)) : GOTO 1020
1000 IF (PT>DT AND DT<=21) OR DT>21 THEN T$="You WIN!  ":
      BANK!=BANK!-BP(PL):BR(PL)=BR(PL)+2*BP(PL) : GOTO 1020
```

```
1010 T$="It's a push " : BR(PL)=BR(PL)+BP(PL)
1020 BP(PL)=0 : GOSUB 1300 : GOSUB 1250 : GOSUB 1260 : GOSUB 1270
1030 IF BR(PL)<=0 THEN BROKE=BROKE+1:LOCATE (PL\2)*8+13,(PL MOD 2)*40+1:
     PRINT "You are BROKE!"
1040 NEXT PL
1050 '
1060 ' check if end of game otherwise play next hand
1070 '
1080 IF BANK!<=0 THEN T$="Whoops!  The house is bankrupt !!!" : GOTO 1170
1090 IF BROKE=NP+1 THEN T$="All players have busted!" : GOTO 1170
1100 R$=" ":T$="Press <SPACE> for next hand":GOSUB 5030
1110 T$=SPACE$(12)
1120 FOR PL=0 TO NP
1130 IF C(PL,0)<>0 THEN GOSUB 1590 : GOSUB 1290 : GOSUB 1300 ' clear hands
1140 NEXT
1150 PL=4 : GOSUB 1590 : LOCATE  4,20 : PRINT SPC(12);
1160 GOTO 430
1170 GOSUB 5120
1180 R$="YN":T$="Want to play some more (Y or N)? " : GOSUB 5030
1190 IF C$="Y" THEN CLS : GOTO 270 ' start another game
1200 IF C$="N" THEN C$=CHR$(27) : GOTO 9050
1210 BEEP : GOTO 1180
1220 '
1230 ' subroutines to display money (bets, bankrolls, etc.) and hand totals
1240 '
1250 LOCATE 4,60 : PRINT USING "$#######"; BANK!; : RETURN
1260 LOCATE (PL\2)*8+12,(PL MOD 2)*40+1 : PRINT USING"$#######";BR(PL); : RETURN
1270 LOCATE (PL\2)*8+7,(PL MOD 2)*40+5  : PRINT USING"$#######";BP(PL); : RETURN
1280 LOCATE (PL\2)*8+13,(PL MOD 2)*40+15: PRINT " ";N$(PL);" "; : RETURN
1290 LOCATE (PL\2)*8+9,(PL MOD 2)*40+1  : PRINT T$; : RETURN
1300 LOCATE (PL\2)*8+10,(PL MOD 2)*40+1 : PRINT T$; : RETURN
1310 '
1320 ' get and display card number CN for player PL, or dealer, PL=4
1330 '
1340 IF P>=NC THEN GOSUB 5250 ' time to shuffle?
1350 P=P+1 : D=D(P) : T=D MOD 100
1360 IF T>10 THEN T=10 ELSE IF T=1 THEN T=11 ' face-card? or ace?
1370 IF PL<4 THEN R=(PL\2)*8+7 : C=(PL MOD 2)*40+10+CN*3 : GOSUB 5340
1380 IF PL=4 THEN R=1 : C=CN*3+34 : GOSUB 5340
1390 RETURN
1400 '
1410 ' add card value T to player's hand
1420 '
1430 C(PL,0)=C(PL,0)+1 : IF T=11 THEN C(PL,2)=1  ' how many cards, ace flag
1440 C(PL,1)=C(PL,1)+T : RETURN  ' hand total
1450 '
1460 ' check and display current hand total
1470 '
1480 T=C(PL,1)
1490 IF C(PL,0)=2 AND T=21 THEN T$="Blackjack" : GOTO 1540
1500 IF C(PL,2)=0 AND T>21 THEN T$="BUSTED!  " : GOTO 1540
1510 IF C(PL,2)=1 AND T>21 THEN T=T-10 : C(PL,1)=T : C(PL,2)=0
1520 IF C(PL,2)=0 THEN T$=STR$(T): GOTO 1540
1530 T$=STR$(T)+" or"+STR$(T-10)
1540 IF PL<4 THEN GOSUB 1290 ELSE LOCATE 4,20 : PRINT T$
1550 RETURN
1560 '
1570 ' clear player PL's cards
```

```
1580 '
1590 CS=(C(PL,0)-1)*3+6 ' how many blanks wide?
1600 IF PL<4 THEN R=(PL\2)*8+6 : C=(PL MOD 2)*40+13
1610 IF PL=4 THEN R=0 : C=37
1620 FOR I=1 TO 6 : LOCATE R+I,C : PRINT SPC(CS); : NEXT
1630 RETURN
5000 '
5010 ' process player responses
5020 '
5030 LOCATE 23,40-LEN(T$)/2 : PRINT T$;
5040 IF R$="" THEN LINE INPUT;"",A$ : GOTO 5080 ' multi-char reply
5050 C$="" : WHILE C$="" : C$=INKEY$ : WEND  ' or single character
5060 IF C$>"Z" THEN C$=CHR$(ASC(C$)-32)      ' shift to upper case
5070 IF INSTR(R$,C$)=0 THEN BEEP : GOTO 5050 ' validate character
5080 LOCATE 23,1 : PRINT SPC(79); : RETURN
5090 '
5100 ' display a message
5110 '
5120 LOCATE 21,1 : PRINT SPC(79);
5130 LOCATE 21,40-LEN(T$)/2 : PRINT T$; : RETURN
5140 '
5150 ' set up deck
5160 '
5170 NC=ND*52-1 : DIM D(NC) : P=-1 ' number cards, deck array and pointer
5180 FOR I=1 TO ND : FOR J=1 TO 4  ' for each deck and suit
5190   FOR K=1 TO 13 : P=P+1 : D(P)=J*100+K : NEXT ' set card numbers
5200 NEXT : NEXT
5210 P=ND*52 : RETURN ' set pointer to force shuffle
5220 '
5230 ' shuffle the cards
5240 '
5250 T$="Excuse me while I shuffle the deck..." : GOSUB 5120
5260 FOR I=1 TO 3 : FOR P=0 TO NC  ' three passes over deck
5270 SWAP D(P),D(INT(RND*NC))      ' exchange each card with a random one
5280 NEXT : NEXT : P=-1            ' set deck pointer when done
5290 T$="":GOSUB 5120:R$=" ":T$="All done. Press <SPACE> to play":GOSUB 5030
5300 RETURN
5310 '
5320 '   Face of 5x6 card
5330 '
5340 COLOR ABS((D\100)<3)*4,7
5350 F1$=MID$("A23456789TJQK",D MOD 100,1):F2$=F1$
5360 IF (D MOD 100)<>10 THEN F1$=F1$+"    ": F2$="    "+F2$
                        ELSE F1$="10  "   : F2$="  10"
5370 LOCATE R+0,C:PRINT CHR$(218);STRING$(4,196);CHR$(191);
5380 LOCATE R+1,C:PRINT CHR$(179);F1$;CHR$(179);
5390 LOCATE R+2,C:PRINT CHR$(179);CHR$(D\100+2);"  ";CHR$(D\100+2)+CHR$(179);
5400 LOCATE R+3,C:PRINT CHR$(179);F2$;CHR$(179);
5410 LOCATE R+4,C:PRINT CHR$(192);STRING$(4,196);CHR$(217);
5420 COLOR 7,0
5430 RETURN
5440 '
5450 '   Back of 5x6 card
5460 '
5470 COLOR 0,7:LOCATE R,C:PRINT CHR$(201);STRING$(4,205);CHR$(187)
5480 FOR I=R+1 TO R+3
5490 LOCATE I,C:PRINT CHR$(186);STRING$(4,176);CHR$(186)
5500 NEXT
5510 LOCATE R+4,C:PRINT CHR$(200);STRING$(4,205);CHR$(188);
5520 COLOR 7,0:RETURN
```

```
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 5.4 BJACK - The card game Blackjack

Variations

Numerous variations are possible in card games. Rules can be relaxed, and some cosmetic changes might be interesting.

1. Add a sound feature in each game. Let the computer play a few "happy" notes when a player wins, and some "sad" notes when he or she loses.

2. When players go bankrupt, let the computer "loan" them money.

3. When the house goes broke, continue the game (even though there will be a negative money amount being displayed).

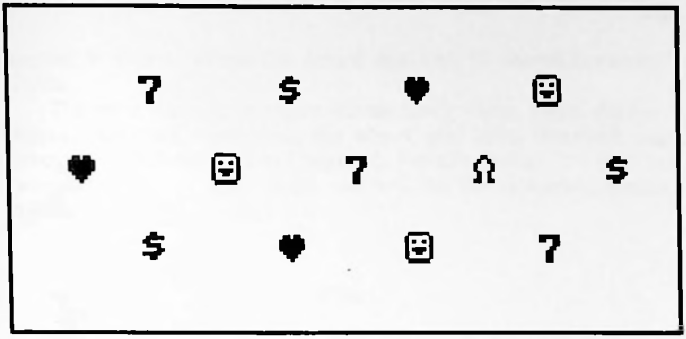
4. Random "messages", using techniques from the word games, could be added to the card games. Let the messages be based on the computer's evaluation of how it plans to bet or some evaluation of how well the player is doing.

5. Add some betting options to the high-card game.

6. Develop an option in the two-card monte poker game to allow a player to draw a new up or down card. This can be handled in a manner similar to how the program handles each player's looking at his down card.

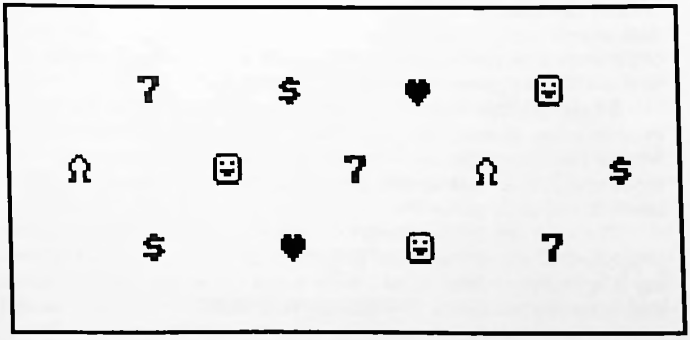
7. Add to blackjack the ability to insure your hand. Splitting pairs into two hands is another possibility, although this is a fairly involved change.

8. Consider some other card games: war—the high card wins the two cards; or stud—five cards with all but one card up and no draw. Baccarat is a type of multiplayer casino card game similar to blackjack where everyone plays the house. The rules are somewhat simpler than blackjack, though.



6

Casino Playtime



Playing casino gambling games on a computer can be loads of fun, and you also do not have the expenses of a gambling junket to Las Vegas or Atlantic City. In our computer-based casino, you can become a "high roller" or big spender and never lose a dime or leave the comforts of your home.

The basis for most gambling games is betting on the outcome of some random event as determined by a device used in the game. For example, roulette in the most basic sense is simply the generation of a random number between 0 and 36. You bet on a single number, or a group of numbers, such as odd, even, black, red; and that the number is between 1 and 18, or between 13 and 24, and so forth.

The game of craps is similar but with two random events: the two dice, with each die generating a random number between 1 and 6. Slots can also be viewed in the same manner, where three random events occur. Selected objects spin on three wheels and eventually stop at some random point. Depending on what specific pattern of objects appear, a payoff is made or not.

Many text-only versions of these games simply allow the player to enter a bet amount, and then tell him the outcome—whether he won or lost. The computer does some bookkeeping on overall winnings and losses, and usually provides some funny dialogue if you break the bank or go broke.

The real skill in programming these games is to incorporate graphics that depict the actual play-by-play action. Actually, showing a spinning roulette wheel is far more fun to play with than a simple message such as "ITS RED 5. YOU WIN!" The same concept

applies to craps, where the actual dice can be shown bouncing about.

The most common of casino games are: roulette, craps, slot machines, blackjack, baccarat, big wheel, and keno. Blackjack has already been developed in Chapter 5. For this chapter, we will develop programs for slots, craps, and roulette. The remaining games are left for you to develop.

Slots

Playing slot machines borders on fanaticism. It is said that when a fire once broke out at a casino, the last people to leave were the slot-machine players (possibly thinking that the flames might finally "warm" up their machines?). Our slot-machine program will let one to four players take turns "pulling back the handle" to spin three wheels and, hopefully, get the big payoff. The program keeps track of how many times each player spins the machine, and each player's winnings. See Figure 6.1 for an example of the game being played.

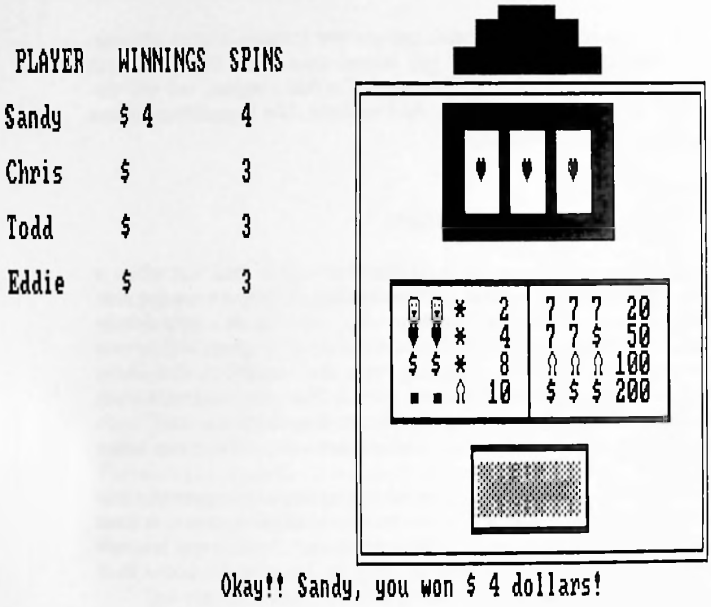
To play the game, you first enter how many players are playing and the players' names, as in the card games. The screen is then cleared and our one-armed bandit is displayed. Each player is asked in turn if he would like a spin or play by printing his name in a message of the following form:

```
Ed, press Esc to exit, <SPACE> to play or 'P' to pass to next player
```

The player must press the space key to play, the P key to let the next player play, or press Esc to end the game. Players can thus compete with each other for the most winnings, or the most winnings with the least number of spins, and so forth.

The program is coded as follows. First, three string arrays, X$, Y$, and Z$, are loaded and the symbols displayed as the machine contained them in DATA statements at the end of the program. The symbols are selected from the PC gaming characters along with the dollar sign, "bar," and the Greek symbol omega, which looks like a small horseshoe. See Figure 6.2 for a list of the symbols, their ASCII character codes, and how many symbols are displayed per wheel.

FIGURE 6.1. Slot machine game example.



Another DATA statement table also contains the payoff symbols and the odds they pay which are loaded into the arrays OD$ and OD, respectively. After this, the program asks the user to indicate the number of players and the players' names, as in the card game programs.

The slot machine is displayed as a series of various extended ASCII graphics characters. A box is drawn for the overall machine; smaller boxes are included in the interior. Solid block characters are used to frame the windows where the three wheels spin; and last, the payoff symbols and dollar amounts are displayed in the lower half of the machine.

A spin of the slots is simply the random display of the "wheel" symbols in the three windows of the machine. The three string array

variables X$, Y$, and Z$, are used with the three variables P0, P1, and P2, which are used as indices.

For some random number of times, the symbols are then printed from each array. The three index variables, P0, P1, and P2, are each incremented some random number of times based on the variables T0, T1, and T2, which are set to the number of times to spin each wheel.

The index variables, P0, P1, and P2 each cycle from 0 to 39 and back to 0 so that the symbols are each displayed in sequence. The symbols are displayed in each window in a column of three characters from each array. This gives the effect of the symbols "rolling by" or spinning on the wheel in the machine.

The three columns gradually slow down and stop "spinning." The overall number of spins is set by the three T variables: T0, T1, and T2. The first column stops changing after T0 spins; the second column stops changing after T1 spins; and the third column stops changing after T2 spins. Once again, this adds to the overall effect of playing with the program: The three wheels slow down one by one just as the actual casino machines do when played.

After a display stops, the three symbols pointed to by the three indices, P0, P1, and P2, are used to determine if a payoff is needed for

FIGURE 6.2. Slot machine character codes.

PAYOFF COMBINATION	AMOUNT	NUMBER OF WAYS	CONTRIBUTION
2	2	5 x 7 x 20 = 700	1400
4	4	5 x 4 x 20 = 400	1600
8	8	3 x 2 x 19 = 114	912
10	10	3 x 2 x 6 = 36	360
7 7 7	20	3 x 3 x 4 = 36	720
7 7 $	50	3 x 3 x 1 = 9	450
Ω Ω Ω	100	1 x 2 x 6 = 12	1200
$ $ $	200	3 x 2 x 1 = 6	1200
		TOTAL 1313	7842

a particular player. A message is displayed if a payoff is made, and that player's winnings are indicated near his name on the screen. The player is then asked to spin again and the game continues.

Each of the three wheels (and arrays) contains twenty symbols in different proportions. The payoff is based on which three symbols show up at the end of a spin. By tallying the number of possible symbol combinations and listing those with a payoff, we can determine the overall percentage or "edge" that the house has over the players.

Figure 6.3 tabulates this percentage for our slot machine program. There are 1313 winning symbol combinations out of a total of 8000 possible combinations. These winning combinations contribute 7842 dollars for the 8000 spins for a 2 percent machine edge.

This means that in the long run, about two cents out of every dollar wagered is kept by the machine. This figure can be easily adjusted and even made positive—for instance, if the house has a losing machine—but, of course, it's "your" house and "your" machine.

FIGURE 6.3. Payoff table for a slot machine game.

CHARACTER	CODE	SYMBOL	NUMBER PER WHEEL (1)	(2)	(3)
Smiley face	1	☺	5	7	3
Big heart	3	♥	5	4	4
The bar	22	▬	3	2	2
Dollar sign	36	$	3	2	1
Lucky seven	55	7	3	3	4
Horseshoe	234	Ω	1	2	6

```
0 ' SLOTS  Slot machine simulation
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  Program simulates a slot machine played by from 1 to 4 players.
50 '
60 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
70 DEFINT A-Z : RANDOMIZE VAL(MID$(TIME$,7,2)+MID$(TIME$,4,2))
80 DIM X$(39),Y$(39),Z$(39),OD$(7),OD(7),N$(3),WINS(3),SPINS(3)
```

```
90 '
100 ' load wheel arrays with symbols (alternating with blank spaces)
110 '
120 FOR J=0 TO 19 : READ C : X$(J*2)=CHR$(C):X$(J*2+1)=" " : NEXT ' wheel 1
130 FOR J=0 TO 19 : READ C : Y$(J*2)=CHR$(C):Y$(J*2+1)=" " : NEXT ' wheel 2
140 FOR J=0 TO 19 : READ C : Z$(J*2)=CHR$(C):Z$(J*2+1)=" " : NEXT ' wheel 3
150 FOR I=0 TO 7
160 READ A,B,C,OD(I) : OD$(I)=CHR$(A)+" "+CHR$(B)+" "+CHR$(C) 'winning combo's
170 NEXT
180 PRINT"SLOTS - Slot machine simulation" : PRINT
190 '
200 ' get number of players and their names
210 '
220 PRINT"How many players (1-4)? ";: NP=0
230 WHILE NP<1 OR NP>4 : NP=VAL(INKEY$) : WEND
240 PRINT NP : NP=NP-1 : PRINT
250 FOR PL=0 TO NP
260   PRINT "Enter player";PL+1;"'s name: ";: LINE INPUT N$
270   IF N$="" THEN BEEP : GOTO 260
280   IF ASC(N$)>96 THEN N$=CHR$(ASC(N$)-32)+MID$(N$,2)
290   N$(PL)=LEFT$(N$,8)
300 NEXT
310 PRINT: PRINT "Press <SPACE> to begin." : WHILE INKEY$<>" ": WEND
320 '
330 ' display the machine
340 '
350 CLS : R=1 : C=35 ' offset for machine (from its upper left corner)
360 COLOR 5,0  ' top of machine
370 LOCATE R,C+13  :PRINT STRING$(5,220);
380 LOCATE R+1,C+11:PRINT STRING$(9,219);
390 LOCATE R+2,C+9 :PRINT STRING$(13,219);
400 COLOR 4,0 ' border about machine
410 LOCATE R+3,C:PRINT CHR$(201);STRING$(29,205);CHR$(187);
420 FOR I=4 TO 19
430 LOCATE R+I,C:PRINT CHR$(186); : LOCATE ,C+30:PRINT CHR$(186);
440 NEXT
450 LOCATE R+20,C:PRINT CHR$(200);STRING$(29,205);CHR$(188);
460 COLOR 2,0 ' windows for wheels
470 LOCATE R+4,C+8:PRINT STRING$(15,219);
480 FOR I=5 TO 7
490 LOCATE R+I,C+8:PRINT STRING$(2,219);
500 LOCATE   ,C+13:PRINT CHR$(219);
510 LOCATE   ,C+17:PRINT CHR$(219);
520 LOCATE   ,C+21:PRINT STRING$(2,219);
530 NEXT I
540 LOCATE R+8,C+8:PRINT STRING$(15,219);
550 COLOR 6,0 ' border for odds
560 LOCATE R+10,C+3
570 PRINT CHR$(218);STRING$(11,196);CHR$(194);STRING$(11,196);CHR$(191);
580 FOR I=11 TO 14
590 LOCATE R+I,C+3:PRINT CHR$(179);SPC(11);CHR$(179);SPC(11);CHR$(179);
600 NEXT
610 LOCATE R+15,C+3
620 PRINT CHR$(192);STRING$(11,196);CHR$(193);STRING$(11,196);CHR$(217);
630 COLOR 7,0 ' display the payoffs
640 FOR I=0 TO 3
650 LOCATE R+I+11,C+5 : PRINT USING "&  ##"; OD$(I);OD(I);
660 LOCATE ,C+17 : PRINT USING "& ###"; OD$(4+I);OD(4+I);
```

```
670 NEXT
680 COLOR 2,5 ' cash box
690 LOCATE R+16,C+10:PRINT CHR$(218);STRING$(9,196);CHR$(191);
700 LOCATE R+17,C+10:PRINT CHR$(179);STRING$(9,177);CHR$(179);
710 LOCATE R+18,C+10:PRINT CHR$(179);STRING$(9,177);CHR$(179);
720 LOCATE R+19,C+10:PRINT CHR$(192);STRING$(9,196);CHR$(217);
730 COLOR 10,0  ' player names, etc.
740 LOCATE 3,5 : PRINT "PLAYER   WINNINGS  SPINS";
750 FOR PL=0 TO NP
760 LOCATE 5+PL*2,4 : PRINT N$(PL);TAB(14);"$";
770 NEXT
780 '
790 ' display initial wheel symbols
800 '
810 COLOR 15,0
820 P0=0 : P1=0 : P2=0          ' pointers to symbol arrays
830 R0=R+5 : R1=R+6 : R2=R+7  ' row of each wheel's symbol
840 C0=C+11: C1=C+15: C2=C+19 '   and its column
850 LOCATE R1,C0:PRINT X$(P0);
860 LOCATE   ,C1:PRINT Y$(P1);
870 LOCATE   ,C2:PRINT Z$(P2);
880 '
890 ' time to play
900 '
910 PL=-1
920 PL=(PL+1) MOD (NP+1) ' get next player
930 LOCATE 24,1 : PRINT SPC(79); ' erase last player message
940 T$=", press Esc to exit, <SPACE> to spin or 'P' to pass to next player
950 T$=N$(PL)+T$ : LOCATE 24,40-LEN(T$)\2 : PRINT T$; ' ask player
960 C$="" : WHILE C$="" : C$=INKEY$ : WEND            ' get reply
970 LOCATE 22,1 : PRINT SPC(79); ' erase status of last spin
980 IF ASC(C$)=27 THEN 9050       ' exit code?
990 IF C$="P" OR C$="p" THEN 920 ' skip to next player?
1000 IF C$<>" " THEN 940          ' not the spin code?
1010 LOCATE 24,1 : PRINT SPC(79); ' erase message to current player
1020 '
1030 ' "spin" the wheels
1040 '
1050 COLOR 15,0
1060 T0=12+INT(RND*40) : IF T0 MOD 2 =1 THEN T0=T0+1 ' rotation time
1070 T1=T0+INT(RND*10)+6 : IF T1 MOD 2 =1 THEN T1=T1+1 ' for each of
1080 T2=T1+INT(RND*10)+10 : IF T2 MOD 2 =1 THEN T2=T2+1  ' the wheels
1090 FOR S=1 TO T2          ' start spinning
1100 IF S<20 OR (S MOD 2) THEN SOUND 500,1
1110 T0=T0-1:IF T0<0 THEN 1160
1120 LOCATE R0,C0 : PRINT X$(P0);  '  display wheel one
1130 LOCATE R1,C0 : PRINT X$((P0+1) MOD 40);
1140 LOCATE R2,C0 : PRINT X$((P0+2) MOD 40);
1150 P0=(P0+1) MOD 40
1160 T1=T1-1:IF T1<0 THEN 1210
1170 LOCATE R0,C1 : PRINT Y$(P1);  '  display wheel two
1180 LOCATE R1,C1 : PRINT Y$((P1+1) MOD 40);
1190 LOCATE R2,C1 : PRINT Y$((P1+2) MOD 40);
1200 P1=(P1+1) MOD 40
1210 IF T0<0 THEN FOR K=1 TO 40 : NEXT K ' pause
1220 IF T1<0 THEN FOR K=1 TO 80 : NEXT K ' pause
1230 LOCATE R0,C2 : PRINT Z$(P2);  '  display wheel three
1240 LOCATE R1,C2 : PRINT Z$((P2+1) MOD 40);
```

```
1250 LOCATE R2,C2 : PRINT Z$((P2+2) MOD 40);
1260 P2=(P2+1) MOD 40
1270 FOR K=1 TO S : NEXT K ' pause loop
1280 NEXT
1290 '
1300 '  pay off for this spin
1310 '
1320 COLOR 10,0 : P$=X$(P0)+" "+Y$(P1)+" "+Z$(P2) : LOCATE 25,24
1330 FOR I=0 TO 7
1340 IF OD$(I)=P$ THEN 1400 ' check exact match
1350 NEXT
1360 FOR I=0 TO 7             ' check "wild-card" combinations
1370 IF RIGHT$(OD$(I),1)="*" AND MID$(P$,1,3)=MID$(OD$(I),1,3) THEN 1400
1380 NEXT
1390 T$="Sorry, "+N$(PL)+", you didn't win anything on this spin." : GOTO 1420
1400 T$="Okay!! "+N$(PL)+", you won $"+STR$(OD(I))+" dollars!"
1410 WINS(PL)=WINS(PL)+OD(I) : LOCATE 5+PL*2,15 : PRINT WINS(PL);
1420 SPINS(PL)=SPINS(PL)+1   : LOCATE 5+PL*2,24 : PRINT SPINS(PL);
1430 LOCATE 22,40-LEN(T$)/2  : PRINT T$; ' display win/lose message
1440 COLOR 7 : GOTO 940  ' continue play
1450 '
1460 ' wheel characters (symbols)
1470 '
1480 DATA 1,22,3,55,1,36,3,234,1,36,3,55,1,22,3,36,1,22,3,55
1490 DATA 1,22,3,1,234,3,1,55,36,22,1,36,3,55,1,234,1,3,55,1
1500 DATA 3,22,55,1,3,234,55,234,22,234,36,234,3,1,234,55,234,3,1,55
1510 '
1520 ' pay offs: 3 symbol codes and odds (42 is "wild card")
1530 '
1540  DATA 1, 1, 42,   2
1550  DATA 3, 3, 42,   4
1560  DATA 36,36,42,   8
1570  DATA 22,22,234,  10
1580  DATA 55,55,55,   20
1590  DATA 55,55,36,   50
1600  DATA 234,234,234,100
1610  DATA 36,36,36,   200
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 6.1 SLOTS - Slot machine simulation

Roulette

Roulette is that elegant game for the rich and distinguished upper-crust gambler. In hopes of defying the odds, the players place their bets on one of many different board positions and watch the bounc-

ing ball fall in a slot in the spinning roulette wheel. Our representation of the game will include a type of spinning wheel with the numbers 0, 00, and 1 to 36. One to four players may play with each taking turns placing their bets on one or more of seven different types of bets. Figure 6.4 illustrates the game of roulette in use.

To play our version of roulette, you must first tell the program how many players are playing, the players' names, and how much money the players have to wager, as in the earlier card game programs. Once play begins, the program queries each player for his bets: the player's name is highlighted, a description of each bet that can be made is displayed in sequence, and the player either bets, skips to the next wager, or exits from betting for the next player to bet.

The program recognizes seven of the many possible roulette bets, leaving out mostly individual numbers and groups of six or less numbers. After a player enters his wager at the bottom of the screen, the wager is posted below the appropriate column in the betting

FIGURE 6.4. Roulette game example.

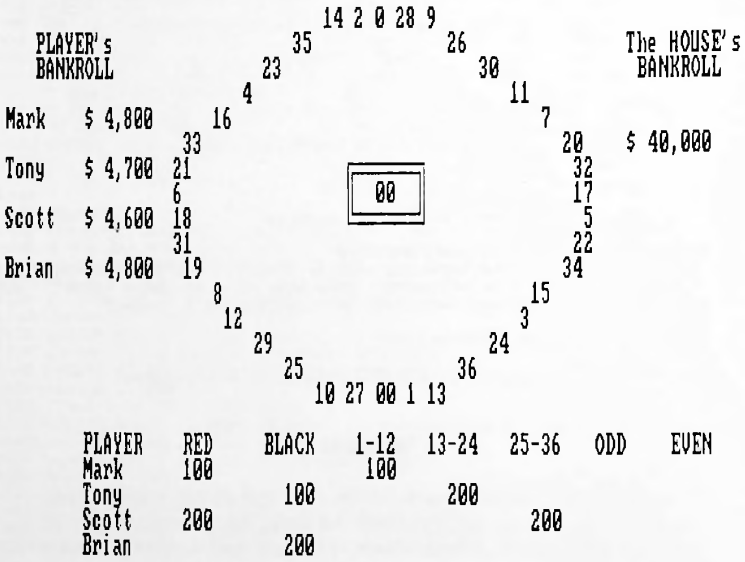


table. If you do not want to make a particular bet, just press the Enter key (this will indicate that no number is entered). When you have no more wagers to make, enter an X to exit from betting. Then the next player will be queried in the same manner. You may also type the word "end" for your bet to be exited from the game altogether.

After all bets are placed, you are asked to "spin the wheel"—and away you go. Numbers are displayed in the middle of the wheel as they are highlighted while spinning around the outside of the wheel. When the wheel stops, the winning number is shown in the middle of the wheel. You then press the space key for the program to highlight the winning bets, and press it again to actually update each player's bankroll and clear the betting area. This allows everybody to see what wagers are winning for who, and so on.

As the wagers are paid off, the program checks to see who has gone broke (a bankroll of 0). Those players that go broke are not allowed to bet any further. If everybody goes broke, the game is over. After the payoffs are made, the game continues with the next round of betting.

The roulette program is designed as follows. The opening sections of code are similar to the card games for gathering player information. The screen is then cleared and the wheel, betting table, player names, and player bankrolls are displayed. Several DATA statement tables are used to aid in this process.

To display the wheel, a DATA statement table at the end of the program contains the coordinates on the screen of each number in the wheel and the number associated with that position. These values are stored in the arrays WR and WC for the wheel's row and column coordinates, and in array WS$ for the wheel number (or symbol) to be used.

The roulette wheel numbers are not listed in sequence from 00, 0, 1, 2, to 36. As the numbers are displayed on the screen with the usual sequence of LOCATE and PRINT statements, they are alternately colored white on black and black on red. For black-and-white monitor users, black on red will appear as black on white. The numbers 0 and 00 are always displayed as green on black.

A second DATA statement table defines the prompts used in placing each player's bets. The table lists the name of the bet, such as RED or BLACK, its column position in the betting table, and its payoff odds. These are also stored in arrays: BD$ for the bet description, BC for the bet's column position in the betting table, and BO for

the bet's payoff odds. Since there are seven different bet types, each of the arrays is indexed by a number from 1 to 7, which is defined in the variable BT.

The program uses the column position for each bet type from the betting table with the player number determining the appropriate row. The descriptions are used as the player is prompted at the bottom of the screen to make his bets. The payoff odds are used when a winning bet is found and the player's bankroll is updated.

Wagers for each player are stored in the two-dimensional array BP with its first index the player number and the second index the type of bet being made, the numbers from 1 to 7 as mentioned above. Before the payoffs are made, it must be determined which of the seven possible wagers were made, and if the wager won based on the final number from the wheel spin. Since each player can make up to seven different wagers, the players will usually win some and lose some at the same time. As the bets are checked, the winning wagers are totaled in the BP array at bet type number 0 for each player.

Each bet type is checked by comparing the number from the wheel N with the criteria for winning that bet. For example, a bet on the first twelve numbers means that N must be from 1 to 12. A bet on ODD means that N must be an odd number, and so on. The only tricky check here is for the red or black number bets. A number's color is determined by its position in the number array, WS$, which is defined by the variable J when the wheel is "spun."

The variable WN is set to whether or not each bet type is a winner or a loser. This is done by setting WN to a true, –1 in BASIC, or false, the number 0. The variable is then used when the player's wagers are checked for this bet type. If WN is true and the player made a bet, then the payoff is made at the appropriate odds for this bet type. Any bets made that did not win for the player are posted to the bankroll of the house.

When a 0 or 00 turns up, all player bets lose. A separate subroutine is used in this case to clear all the bets in the BP array and update the house bankroll.

As the payoffs are made, their entries in the betting table are reprinted black on white to highlight the winners. After all bets are checked, the winning total from the BP array is added to the player's overall bankroll. The betting table is then cleared of all bets, and the total value in the BP array is reset to 0.

The listing for the roulette game is contained in Program 6.2. To change the types of bets or even the odds (remember, it is your house), just change the entries in the previously described tables. A column for individual wheel number bets could also be added. The betting queries for each player would then have to first ask the number being bet on and then the wager.

```
0 ' ROUL   Roulette wheel gambling
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  Up to four players play roulette.
50 '
60 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
70 DEFINT A-Z : DEFSNG B : RANDOMIZE VAL(LEFT$(TIME$,2)+MID$(TIME$,4,2))
80 DIM BR(3), BP(3,7), N$(3)    ' bankroll, player bets and names
90 DIM WR(37),WC(37),  WS$(37) ' wheel row, column, symbol
100 DIM BD$(7),BC(7),  BO(7)    ' bet description, column, odds
110 FOR I=0 TO 37 : READ WR(I),WC(I),WS$(I) : NEXT  ' load wheel description
120 FOR I=1 TO 7  : READ BC(I),BO(I),BD$(I) : NEXT  ' load bet parameters
130 PRINT"ROUL - Roulette Wheel Gambling" : PRINT
140 '
150 '  Get number of players, their names and bankroll
160 '
170 PRINT"How many players (1-4)? "; : NP=0
180 WHILE NP<1 OR NP>4 : NP=VAL(INKEY$) : WEND
190 PRINT NP : NP=NP-1 : PRINT
200 FOR PL=0 TO NP
210   PRINT "Enter player";PL+1;"'s name: "; : LINE INPUT N$
220   IF N$="" THEN BEEP : GOTO 210
230   IF ASC(N$)>96 THEN N$=CHR$(ASC(N$)-32)+MID$(N$,2)
240   N$(PL)=LEFT$(N$,8)
250 NEXT
260 LINE INPUT "What is each player's bankroll: ",N$
270 B=VAL(N$) : IF B=<1 OR B>100000! THEN BEEP : GOTO 260
280 FOR PL=0 TO NP : BR(PL)=B : NEXT
290 BANK=2*B*(NP+1)  ' house bankroll
300 PRINT: PRINT "Press <SPACE> to begin." : WHILE INKEY$<>" ": WEND
310 '
320 '  create overall screen display
330 '
340 CLS
350 LOCATE 2,6:PRINT"PLAYER's" : LOCATE 2,64:PRINT"The HOUSE's"
360 LOCATE 3,6:PRINT"BANKROLL" : LOCATE 3,65:PRINT"BANKROLL"
370 FOR PL=0 TO NP
380 LOCATE 5+PL*2,3 : PRINT N$(PL); : GOSUB 1340 ' names and bankrolls
390 NEXT
400 GOSUB 1330 : LOCATE 18,11
410 PRINT"PLAYER    RED     BLACK    1-12   13-24   25-36   ODD    EVEN";
420 FOR PL=0 TO NP : GOSUB 1350 : NEXT ' print names by bets
430 FOR I=0 TO 37
440 IF (I MOD 2) THEN COLOR 4,0 ELSE COLOR 0,7 ' red or black
450 IF VAL(WS$(I))=0 THEN COLOR 2,0  ' green for 0 and 00
460 LOCATE WR(I),WC(I):PRINT WS$(I); ' display the wheel
470 NEXT : COLOR 7,0
```

```
480 LOCATE 7,37 : PRINT CHR$(201);STRING$(6,205);CHR$(187);
490 LOCATE 8,37 : PRINT CHR$(186); : LOCATE ,44 : PRINT CHR$(186);
500 LOCATE 9,37 : PRINT CHR$(200);STRING$(6,205);CHR$(188);
510 '
520 ' get player bets
530 '
540 FOR PL=0 TO NP
550 IF BR(PL)<=0 THEN 680  ' does player have any money?
560 COLOR 0,7 : GOSUB 1350 : COLOR 7,0  ' highlight player name
570   FOR B=1 TO 7
580   T$=N$(PL)+", want to bet on "+BD$(B)+" (Press Enter for no bet)?"
590   GOSUB 1390 : IF A$="" THEN 660 ' skip this bet
600   IF A$="X" OR A$="x" THEN 670   ' no more betting
610   IF A$="END" OR A$="end" OR A$="End" THEN 1140 ' end game
620   BET=VAL(A$) : IF BET<1 OR BET>BR(PL) THEN BEEP : GOTO 580
630   BP(PL,B)=BET : BR(PL)=BR(PL)-BET : GOSUB 1340
640   LOCATE 19+PL,BC(B) : PRINT USING "#####";BET;
650   IF BR(PL)<=0 THEN B=7 ' player bet everything
660   NEXT B
670 GOSUB 1350 ' clear player name
680 NEXT PL
690 '
700 '  spin the wheel
710 '
720 T$="PRESS <SPACE> TO SPIN THE WHEEL": GOSUB 1440
730 FOR I=INT(RND*38) TO INT(90+RND*38)  ' random start/end
740  J=I MOD 38 : N$=WS$(J)                ' get a number
750  LOCATE WR(J),WC(J) : PRINT SPC(LEN(N$));  ' clear it off wheel
760  IF (J MOD 2) THEN COLOR 4,0 ELSE COLOR 0,7 ' set red, black
770  IF VAL(WS$(J))=0 THEN COLOR 2,0            ' or green for 0 and 00
780  LOCATE 8,39:PRINT USING "\  \";" "+N$+" ";  ' display in middle
790  IF J MOD 2 THEN SOUND 600,1 ELSE SOUND 900,1 ' clicking sound
800  FOR D=0 TO (I\8)^1.75: NEXT D  ' delay
810  LOCATE WR(J),WC(J) : PRINT N$; ' re-display the number
820 NEXT
830 '
840 ' check for the winning bets
850 '
860 COLOR 7,0 : N=VAL(N$) ' N is final number
870 FOR PL=0 TO NP
880 IF N=0 THEN GOSUB 1260 : GOTO 960  ' all bets lose on 0 or 00
890 B=1 : WN=(J MOD 2)=1 : GOSUB 1180  ' RED
900 B=2 : WN=(J MOD 2)=0 : GOSUB 1180  ' BLACK
910 B=3 : WN=N>0  AND N<13:GOSUB 1180  ' 1st 12
920 B=4 : WN=N>12 AND N<25:GOSUB 1180  ' 2nd 12
930 B=5 : WN=N>24 AND N<37:GOSUB 1180  ' 3rd 12
940 B=6 : WN=(N MOD 2)=1 : GOSUB 1180  ' Odd
950 B=7 : WN=(N MOD 2)=0 : GOSUB 1180  ' Even
960 NEXT
970 '
980 ' pay off bets, see who's broke, clear screen and play some more
990 '
1000 T$="Press <SPACE> to pay off bets": GOSUB 1440: BROKE=0
1010 FOR PL=0 TO NP
1020 LOCATE 19+PL,19 : PRINT SPC(60);  ' clear bets
1030 BR(PL)=BR(PL)+BP(PL,0) : BP(PL,0)=0 : GOSUB 1340 ' payoff
1040 IF BR(PL)=0 THEN BROKE=BROKE+1
1050 NEXT : GOSUB 1330 : IF BANK<=0 THEN 1080
```

```
1060 IF BROKE<NP+1 THEN 540
1070 LOCATE 21,24 : PRINT"Everybody's broke!  Game is over." : GOTO 1090
1080 LOCATE 21,24 : PRINT"Game over. You broke the bank!"
1090 LOCATE 23,24 : PRINT "Want to play some more (Y/N)?"
1100 C$="" : WHILE C$="" : C$=INKEY$ : WEND
1110 IF C$>"Z" THEN C$=CHR$(ASC(C$)-32)
1120 IF C$="Y" THEN CLS : GOTO 260
1130 IF C$="N" THEN C$=CHR$(27) : GOTO 9050
1140 BEEP : GOTO 1090
1150 '
1160 ' calculate payoff, if any, for player PL
1170 '
1180 IF BP(PL,B)=0 THEN RETURN
1190 IF BP(PL,B)>0 AND NOT WN THEN BANK=BANK+BP(PL,B) : GOTO 1220
1200 BP=INT(BP(PL,B)*BO(B)) : BP(PL,0)=BP(PL,0)+BP(PL,B)+BP : BANK=BANK-BP
1210 COLOR 0,7 : LOCATE 19+PL,BC(B) : PRINT USING "#####";BP(PL,B); : COLOR 7,0
1220 BP(PL,B)=0 : RETURN
1230 '
1240 ' clear losing bets for 0 and 00
1250 '
1260 FOR B=1 TO 7
1270 IF BP(PL,B)>0 THEN BANK=BANK+BP(PL,B) : BP(PL,B)=0
1280 NEXT
1290 RETURN
1300 '
1310 ' subroutines to display money
1320 '
1330 LOCATE 6,64       : PRINT USING "$###,###";BANK;  : RETURN
1340 LOCATE 5+PL*2,11 : PRINT USING "$###,###";BR(PL); : RETURN
1350 LOCATE 19+PL,11  : PRINT N$(PL); :  RETURN
1360 '
1370 ' process player responses for bets
1380 '
1390 LOCATE 23,40-LEN(T$)/2 : PRINT T$; : LINE INPUT;" ",A$
1400 LOCATE 23,1 : PRINT SPC(79); : RETURN
1410 '
1420 ' wait for <SPACE>
1430 '
1440 LOCATE 23,40-LEN(T$)\2: PRINT T$;
1450 WHILE INKEY$<>" " : WEND
1460 LOCATE 23,1 : PRINT SPC(79); : RETURN
1470 '
1480 '  Row, Column coordinates for positions in roulette wheel
1490 '
1500 DATA  1,35,14,   1,38,2,    1,40,0,    1,42,28, 1,45,9
1510 DATA  2,47,26,   3,50,30,   4,53,11,   5,56,7
1520 DATA  6,58,20,   7,59,32,   8,59,17,   9,60,5,   10,59,22
1530 DATA 11,58,34,  12,55,15,  13,54,3,   14,51,24, 15,48,36
1540 DATA 16,45,13,  16,43,1,   16,40,00,  16,37,27, 16,34,10
1550 DATA 15,31,25,  14,28,29,  13,25,12,  12,24,8,   11,21,19
1560 DATA 10,20,31,   9,20,18,   8,20,6,    7,20,21,   6,21,33
1570 DATA  5,24,16,   4,27,4,    3,29,23,   2,32,35
1580 '
1590 ' betting: column location on screen, payoff odds and description
1600 '
1610 DATA 19, 1, RED
1620 DATA 29, 1, BLACK
1630 DATA 37, 2, 1st 12 (1 to 12)
```

```
1640 DATA 45, 2, 2nd 12 (13 to 24)
1650 DATA 53, 2, 3rd 12 (25 to 36)
1660 DATA 59, 1, ODD
1670 DATA 67, 1, EVEN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 6.2 ROUL - Roulette game

Craps

The game of craps is fast-paced, wild, and somewhat physical: The players throw down their bets, grab the dice, whisper a few incantations, and fling the dice across the table. Like roulette, craps has many different types of bets that each player can make. Additionally, the players take turns "owning" and throwing the dice, betting that they pass or don't pass. The rest of the players bet on other side bets. Our version of craps will provide three such additional bets so more than one player can participate in the game at the same time. See Figure 6.5 for an example of the game being played.

For the player with the dice, a Pass bet is as follows. The player bets that he will roll a 7 or 11 on the first roll. If neither a 7 nor an 11 is rolled, then the number that *is* rolled becomes his "point," and he continues to roll until he makes his point or rolls a 2 or an 11, which causes him to lose.

A Don't Pass bet is the opposite of a Pass bet. The player with the dice expects to roll numbers other than a 7 or 11, and if he rolls for his point, he hopes he doesn't get it. Both the Pass and Don't Pass bets pay off at even odds; that is, you win an amount equal to what you wagered.

The side bets in our game are Play the Field, Any 7, and Any Craps. These bets can be made before any roll of the dice by any player. Play-the-Field wagers are bets that the dice will turn up a 2, 3, 4, 9, 10, 11, or 12. It pays at even odds except for the 2 or 12, which pay at 2 to 1. The Any 7 wager is a bet that a 7 will come up on the next roll, and pays off at odds of 4 to 1. The Any Craps wager is a bet that a 2, 3, or 12 will turn up next; it pays at odds of 7 to 1.

Playing our craps game is similar to the roulette program. After identifying the players and their bankrolls, the game begins. The first player has the dice and is asked to bet that they Pass or Don't Pass. He must either make a bet for one or the other, or pass the dice on to the next player. (You may also enter the word "end" at this point to exit from the game altogether.)

After the Pass/Don't Pass bet is made, all players are asked in turn if they want to make any of the three side bets. After all bets are down, you press the space key to roll the dice. Note that since there are only three other bets to make, you just press Enter to skip over a bet you do not want to make. You cannot enter X to skip to the next player, as in roulette.

After the dice stop rolling, the payoffs are made. If the player with the dice wins, then he or she is allowed to roll again; if the first player does not win, the next player is asked to wager on a Pass or Don't Pass. If the player has to make a point, then he continues to roll

FIGURE 6.5. Craps game example.

```
PLAYER's BANKROLL                                    HOUSE BANKROLL

Jim      $  3,600
Randy    $  3,300          ┌─────┐   ┌─────┐          $ 43,150
Bill     $  3,100          │  0  │   │ 0  0│
Howie    $  4,150          └─────┘   │ 0  0│
                                     └─────┘

                    Rolled 1,4 for a 5

                    Your point, Randy, is 5

                    DON'T   PLAY THE      ANY      ANY
PLAYER      PASS    PASS      FIELD       7       CRAPS
Jim                                      500
Randy       500             100         100       100
Bill                        500         200       100
Howie                       500         100

                    Time to place side bets

                    Howie, want to bet on ANY CRAPS?
```

until the point is made, or he loses. In either case, all players are given an opportunity to make side bets before each roll.

The program logic for craps is straight-forward, the main concern being to keep track of which player has the dice, whether he bets Pass or Don't Pass, and which roll of the dice it is. The roll of the dice is important because the second roll relates to dice values, depending on how the player bet on the Pass line (that is, if the player made the point, or not).

Each column in the bet table displayed on the screen is ten spaces apart. There are only three columns for each player's possible side bets (other than the Pass/Don't Pass bet columns). Thus, there is no need for a DATA statement table (as in the roulette game) to control the betting sequence with betting descriptions, their screen positions, and payoff odds.

Wagers are stored for each player in a two-dimensional array BP, structured as in the roulette game. The bet type ranges from 1 to 4, with index 1 being the Pass/Don't Pass wager, and indices 2, 3, and 4 the side bet wagers. The variable P is set to 0 for the player with the dice to indicate that a Pass bet has been made, or 1 for Don't Pass.

Displaying and rolling the dice is done with two subroutines. First, a pair of boxes is drawn with the PC's business-line graphics characters (listed in Figure 2.1 on page 28). Then another subroutine displays the "spots," or number patterns on each die, using the letter o. Before a die is displayed, the interior of the box is erased by printing blanks. A random number between 1 and 6 is calculated and a GOTO line based on that number is made. That line then displays the appropriate number of o's for the number.

The process is repeated a few times for effect, along with a few sounds played at the same time. After a few moments of this, the last number displayed or "rolled" for each die is returned to the main program for determining the winners. The code can be easily extracted and used in other dice games. This is done in the backgammon game in Chapter 7. Program 6.3 is the complete listing of the craps game.

```
0 '  CRAPS   Dice gambling game
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  Roll the dice and play the game of craps.
50 '
60 SCREEN 0:WIDTH 80:COLOR 7,0:KEY OFF:CLS
70 RANDOMIZE VAL(LEFT$(TIME$,2)+MID$(TIME$,4,2))
```

```
80 DEFINT A-Z : DEFSNG B  : DIM BR(3),BP(3,5),N$(3)
90 PRINT"CRAPS  Casino Dice Game" : PRINT
100 '
110 '  Get number of players, their names and bankroll
120 '
130 PRINT"How many players (1-4)? "; : NP=0
140 WHILE NP<1 OR NP>4 : NP=VAL(INKEY$) : WEND
150 PRINT NP : NP=NP-1 : PRINT
160 FOR PL=0 TO NP
170   PRINT "Enter player";PL+1;"'s name: "; : LINE INPUT N$
180   IF N$="" THEN BEEP : GOTO 170
190   IF ASC(N$)>96 THEN N$=CHR$(ASC(N$)-32)+MID$(N$,2)
200   N$(PL)=LEFT$(N$,8)
210 NEXT
220 LINE INPUT"What is each player's bankroll: ",N$
230 B=VAL(N$) : IF B<1 OR B>100000! THEN BEEP : GOTO 220
240 FOR PL=0 TO NP : BR(PL)=B : NEXT
250 BANK=2*B*(NP+1)   ' house bankroll
260 PD=-1 : PP=0      ' initialize player with dice, and previous player
270 '
280 '  create overall screen display
290 '
300 CLS : R=3 : C=28 : GOSUB 1670 : C=42 : GOSUB 1670
310 LOCATE 2,7 :PRINT"PLAYER's BANKROLL";
320 LOCATE ,55 :PRINT"HOUSE BANKROLL";
330 LOCATE 13,9:PRINT"                              DON'T   PLAY THE       ANY       ANY"
340 LOCATE 14,9:PRINT"PLAYER       PASS              PASS     FIELD         7       CRAPS"
350 FOR PL=0 TO NP
360 LOCATE PL+4,6 : PRINT N$(PL); : GOSUB 1750 : GOSUB 1760
370 NEXT : GOSUB 1740
380 '
390 '  get bet of player with the dice
400 '
410 LOCATE 15+PP,20 : PRINT SPC(19);   ' erase previous player's bet,
420 PD=(PD+1) MOD (NP+1) : PP=PD       ' next player
430 LOCATE 15+PD,20 : PRINT SPC(19);   ' erase last bet
440 LOCATE 9,29  : PRINT SPC(40);      '    and point message
450 LOCATE 11,29 : PRINT SPC(40);      '    and dice total
460 T$="Place your bet, "+N$(PD)+". You have the dice." : GOSUB 1850
470 T$="Enter bet that you PASS (press Enter to not bet):": GOSUB 1800
480 IF A$="END" OR A$="end" OR A$="End" THEN 1020
490 IF A$="X" OR A$="x" THEN 410 ELSE IF A$="" THEN 520
500 BET=VAL(A$) : IF BET<1 OR BET>BR(PD) THEN BEEP : GOTO 460
510 P=1 : GOTO 570 ' PASS flag
520 T$="Enter bet that you DON'T PASS:": GOSUB 1800
530 IF A$="END" OR A$="end" OR A$="End" THEN 1020
540 IF A$="X" OR A$="x" THEN 410 ELSE IF A$="" THEN BEEP : GOTO 460
550 BET=VAL(A$) : IF BET<1 OR BET>BR(PD) THEN BEEP : GOTO 520
560 P=2 ' DON'T PASS flag
570 BT=P : PL=PD : GOSUB 1340
580 GOSUB 1180  ' other bets
590 '
600 ' initial roll of dice
610 '
620 T$="Press <SPACE> to roll dice": GOSUB 1900
630 RL=1 : GOSUB 1460 : PW=0
640 IF (DN=7 OR DN=11) AND P=1 THEN PW=1          ' PASS win
650 IF (DN=2 OR DN=3 ) AND P=2 THEN PW=1          ' DON'T PASS win
660 IF (DN=2 OR DN=3 OR DN=12) AND P=1 THEN PW=2 ' PASS loss
```

```
670 IF (DN=7 OR DN=11) AND P=2 THEN PW=2          ' DON'T PASS loss
680 IF (DN=12 AND P=2) THEN PW=3                  ' DON'T PASS draw
690 IF PW=1 THEN T$=N$(PD)+", you win!!!" : GOSUB 1850
700 IF PW=1 THEN BT=P : PL=PD : OD=1 : GOSUB 1400 ' payoff win
710 IF PW=2 THEN T$=N$(PD)+", you lose." : GOSUB 1850
720 IF PW=2 THEN BANK=BANK+BP(PD,P) : BP(PD,P)=0  ' bank gets loss
730 IF PW=3 THEN T$="Its a draw "+N$(PD) : GOSUB 1850
740 IF RL=1 AND PW=0  THEN PT=DN
750 IF PW=0 THEN LOCATE 11,29 : PRINT "Your point, ";N$(PD);", is";PT
760 '
770 ' check other player bets
780 '
790 FOR PL=0 TO NP
800 IF BP(PL,3)>0 AND (DN=2 OR DN=12) THEN OD=2 : BT=3 : GOSUB 1400 : GOTO 820
810 IF BP(PL,3)>0 AND (DN<5 OR DN>8) THEN OD=1 : BT=3 : GOSUB 1400
820 IF BP(PL,4)>0 AND DN=7 THEN OD=4 : BT=4 : GOSUB 1400
830 IF BP(PL,5)>0 AND (DN=2 OR DN=3 OR DN=12) THEN OD=7 : BT=5 : GOSUB 1400
840 FOR BT=3 TO 5 : BANK=BANK+BP(PL,BT) : BP(PL,BT)=0 : NEXT ' losing bets
850 NEXT
860 '
870 ' pay off bets, check who's broke and continue next roll
880 '
890 T$="Press <SPACE> to pay off bets...": GOSUB 1900: BROKE=0
900 FOR PL=0 TO NP
910 IF BP(PL,0)>0 THEN BR(PL)=BR(PL)+BP(PL,0) : GOSUB 1750
920 BP(PL,0)=0 : LOCATE 15+PL,39 : PRINT SPC(40);  ' clear bets
930 IF BR(PL)=0 THEN BROKE=BROKE+1
940 NEXT : GOSUB 1740 : IF BANK<=0 THEN 970
950 IF BROKE<>NP+1 OR (BROKE=NP+1 AND PW<>2) THEN 1030 ' everybody broke?
960 LOCATE 21,24 : PRINT"Everbody's broke!  Game is over." : GOTO 980
970 LOCATE 21,24 : PRINT"Game over!  You broke the bank!!"
980 LOCATE 23,24 : PRINT"Do you want to play again (Y/N)?"
990 C$="" : WHILE C$="" : C$=INKEY$ : WEND
1000 IF C$>"Z" THEN C$=CHR$(ASC(C$)-32)
1010 IF C$="Y" THEN CLS:GOTO 220 ELSE IF C$="N" THEN C$=CHR$(27) : GOTO 9050
1020 BEEP : GOTO 980
1030 IF PW=1 OR (PW=3 AND RL=1) THEN 430 ' player PD won/draw and continues
1040 IF PW=2 THEN 410 ' player PD lost, next player gets dice
1050 '
1060 ' players bet again and player PD rolls for point
1070 '
1080 GOSUB 1180 : LOCATE 9,29 : PRINT SPC(40);
1090 T$="Press <SPACE> to roll 'em again": GOSUB 1900
1100 RL=2 : GOSUB 1460 : PW=0
1110 IF (DN=PT AND P=1) OR (DN=7 AND P=2) THEN PW=1   ' PASS/DON'T PASS win
1120 IF (DN=7 AND P=1) OR (DN<>12 AND P=2) THEN PW=2 ' PASS/DON'T PASS loss
1130 IF (DN=12 AND P=2) THEN PW=3                     ' DON'T PASS draw
1140 GOTO 690
1150 '
1160 ' get other bets
1170 '
1180 T$="Time to place side bets" : GOSUB 1850
1190 FOR PL=0 TO NP
1200 IF BR(PL)=0 THEN 1260 ' player broke?
1210 COLOR 0,7 : GOSUB 1760 : COLOR 7,0 ' highlight player name
1220 T$=N$(PL)+", want to PLAY the FIELD?" : BT=3 : GOSUB 1310
1230 T$=N$(PL)+", bet on ANY 7?" : BT=4 : GOSUB 1310
1240 T$=N$(PL)+", want to bet on ANY CRAPS?" : BT=5 : GOSUB 1310
```

```
1250 GOSUB 1760  ' clear player name
1260 NEXT
1270 T$="" : GOSUB 1850 : RETURN
1280 '
1290 ' process other bets
1300 '
1310 COLOR 0,7 : LOCATE 15+PL,10+BT*10 : PRINT SPC(5); : COLOR 7,0
1320 GOSUB 1800 : IF A$="" THEN LOCATE 15+PL,10+BT*10 : PRINT SPC(6); : RETURN
1330 BET=VAL(A$) : IF LEN(A$)>5 OR BET<1 OR BET>BR(PL) THEN BEEP : GOTO 1320
1340 BP(PL,BT)=BET : BR(PL)=BR(PL)-BET : GOSUB 1750
1350 LOCATE 15+PL,10+BT*10 : PRINT USING "######";BP(PL,BT);
1360 RETURN
1370 '
1380 ' highlight and pay-off bet BT for player PL at odds OD
1390 '
1400 BP(PL,0)=BP(PL,0)+BP(PL,BT)*(OD+1)
1410 COLOR 0,7 : GOSUB 1350 : COLOR 7,0
1420 BANK=BANK-BP(PL,BT) : BP(PL,BT)=0 : RETURN
1430 '
1440 ' roll 'em
1450 '
1460 FOR T=1 TO RND*8+8
1470 R=4:C=30:GOSUB 1560:D1=D:C=44:GOSUB 1560:D2=D
1480 FOR D=0 TO T : NEXT D : SOUND 1200,1
1490 NEXT
1500 DN=D1+D2
1510 LOCATE 9,29 : PRINT USING "Rolled  #_,#  for a ##";D1;D2;DN;
1520 RETURN
1530 '
1540 ' roll (display) a die
1550 '
1560 FOR I=0 TO 2:LOCATE R+I,C:PRINT"      ";:NEXT
1570 D=INT(RND*6)+1 : ON D GOTO 1580,1590,1600,1610,1620,1630
1580 LOCATE R+1,C+2:PRINT"o";:RETURN
1590 LOCATE R,C:PRINT"o";:LOCATE R+2,C+4:PRINT"o";:RETURN
1600 LOCATE R,C:PRINT"o";:LOCATE R+1,C+2:PRINT"o";:LOCATE R+2,C+4:PRINT"o";:RETU
RN
1610 LOCATE R,C:PRINT"o    o";:LOCATE R+2,C:PRINT"o    o";:RETURN
1620 LOCATE R,C:PRINT"o    o";:LOCATE R+1,C+2:PRINT"o";:LOCATE R+2,C:PRINT"o    o"
;:RETURN
1630 LOCATE R,C:PRINT"o    o";:LOCATE R+1,C:PRINT"o    o";:LOCATE R+2,C:PRINT"o
o";:RETURN
1640 '
1650 ' print 5x9 box
1660 '
1670 LOCATE R,C : PRINT CHR$(218);STRING$(7,196);CHR$(191);
1680 FOR I=1 TO 3:LOCATE R+I,C:PRINT CHR$(179);SPACE$(7);CHR$(179); : NEXT
1690 LOCATE R+4,C : PRINT CHR$(192);STRING$(7,196);CHR$(217);
1700 RETURN
1710 '
1720 '  display house or player PL's bankroll, or player name
1730 '
1740 LOCATE 4,58     : PRINT USING "$###,###";BANK; : RETURN
1750 LOCATE PL+4,16 : PRINT USING "$###,###";BR(PL); : RETURN
1760 LOCATE PL+15,9 : PRINT N$(PL); : RETURN
1770 '
1780 ' process player responses
1790 '
```

```
1800 LOCATE 23,40-LEN(T$)/2 : PRINT T$; : LINE INPUT;" ",A$
1810 LOCATE 23,1 : PRINT SPC(79); : RETURN
1820 '
1830 ' display a message
1840 '
1850 LOCATE 21,1 : PRINT SPC(79);
1860 LOCATE 21,40-LEN(T$)/2 : PRINT T$; : RETURN
1870 '
1880 ' wait for <SPACE>
1890 '
1900 LOCATE 23,40-LEN(T$)/2: PRINT T$;
1910 IF INKEY$<>" " THEN 1910
1920 LOCATE 23,1 : PRINT SPC(79); : RETURN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 6.3 CRAPS - Casino dice game

Variations

Our gambling games have covered the bulk of the variations that can be played in these games. Most enhancements are concerned with cosmetics.

1. Convert the slot machine game to a 40-column-width mode. The code as given can be modified by first changing the starting offset for the machine display, then narrowing the width of the machine. Making room for the bets is another concern—they could be displayed below the machine.

2. A jackpot with flashing characters at the top of the slot machine might be an interesting addition. Adjust the odds so it pays off fairly often. Perhaps a flashing character and beeping-tone sequence can be added to all wins.

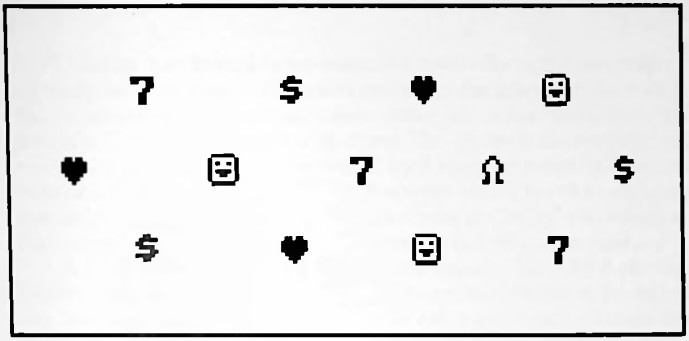
3. More color can easily be added to the slot machine. For instance, its interior might be colored red or green.

4. The roulette and craps game have room for additional bets.

5. Additional color is possible in both the roulette and craps games. The interiors of the wheel and dice might be changed to white or a color.

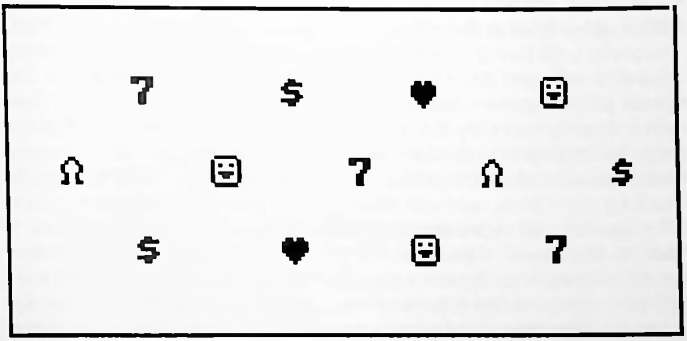
6. All three games could be converted to a graphics mode, although the craps game appears to have the least to gain with this variation. A graphics-mode roulette wheel could then be a true round circle; the slot machine could be drawn with more intricate detail.

7. Combine the games into one main program and keep track of each player's winnings (and losses) in a disk file. Let the players change from one game to another without changing their bankrolls. After the games are played, these amounts could be updated on the disk file for an ongoing tally of how well your players have done. Rename the game "Casino Party."



7

Board Games



Say "board games," and one might imagine games ranging from Monopoly and Trivial Pursuit to chess and checkers. There are a host of such games. All these games can lend themselves to a computer-based representation with varying degrees of difficulty and success. At the low end of the scale, the computer might simply display the board for the player or players to use. The computer might also act as a referee. At the other end of the scale, the computer can become your opponent, a fellow player, and even a coach or teacher. Many electronic chess games will suggest moves you might make, grade the move you ultimately make, as well as play against you or even itself.

DEVELOPING BOARD GAMES

Three issues arise in developing a board game on a computer. First, the game must be represented both *graphically* on the screen for the player to view and *internally* for the program to process. Second, the game play must be coded in such a manner that the player can easily interact and play the game. Last, some degree of intelligence must be developed in the computer's play of the game: This is often a major issue, leading to topics such as heuristics, searching and backtracking algorithms, and ultimately artificial intelligence techniques.

The external representation of the game is its graphical depiction on the screen. This is usually a matter of coding the proper graphics statements for the board and the pieces that are used during play, based on the nature of the game. It should clearly show the player the location of the game's pieces. The internal representation of the game is a different matter.

Usually, the board is represented internally to the program as an array with entries in the array coded for the pieces on the board. This internal representation often differs in many ways from the screen's. Consider the game of chess. To represent the chess board you could use a two dimensional 8 by 8 numeric array to store the location of the board's pieces. On the other hand, to aid in ensuring that only correct numeric moves are made and to help in selecting the comptuer's moves, a 12 by 12 array is actually more useful.

A border of two rows and columns around the 8 by 8 playing area can be coded so that moves of pieces at the edge of the board can be more easily processed. These extra rows and columns are usually coded in such a manner that no move is allowed for them. Thus, a piece at the edge of the board need not be handled any differently from a piece at the interior of the board.

Player interaction in board games can be designed in many different ways. Two general approaches can be considered. First, each board position can be represented and shown on the screen with some type of code for the row and column of the game's pieces. These codes are then entered into the game to indicate the player's move. Converting the codes into the actual internal position being represented is usually a fairly simple matter.

An alternative to this is to use some type of cursor movement to choose pieces on the board. For example, after the space bar is pressed, the program might highlight successive positions on the board where the player's pieces are located. To indicate that a piece has been selected, another key might be pressed—such as Enter. The cursor control keys can also be used in a manner similar to the space bar. Recall that the concentration game in Chapter 2 used cursor control keys for selecting its cards.

This form of interaction is generally better than the coded text entry described earlier. The actual pieces being selected by the player are more easily discerned when they are highlighted. The entry of codes to represent positions can be confusing if it is not designed carefully, although once learned, codes may be faster than moving a cursor about.

Providing the program with intelligent game play is usually the most difficult part of programming board games. In general, a brute-force technique of finding all possible moves and all possible replies quickly falls apart because of the enormous number of choices that must be considered. Instead, some form of judgment must be coded within the program to selectively consider which moves should be

analyzed and to what extent: This is known as selective pruning. The main concern is how to evaluate which moves should be considered and which should not, and to what degree. Indeed, much of artificial intelligence research concerns itself with how judgments or rules are made and how a program might "learn" them itself.

Our two sample board game programs use a fairly simple move-selection technique. Every possible move that the computer can make for a given board arrangement will be assigned some numeric value that rates the value of the move. The best move, which is the one the computer will make, will be the one with the highest value.

The two games developed in this chapter are checkers and backgammon.

Checkers

It is assumed that everyone is familiar with the game of checkers. Play consists of two players, each taking turns moving one of their twelve pieces about a board. The object of the game is to capture all of your opponent's pieces before he or she captures yours. In our game, your opponent is the computer.

Either you or the computer may go first. The computer has the black double-line pieces and you have the solid black pieces. Figure 7.1 illustrates the graphical form that the pieces take in the game. Each player's twelve pieces are positioned on alternate squares on the first three rows of his side of the board.

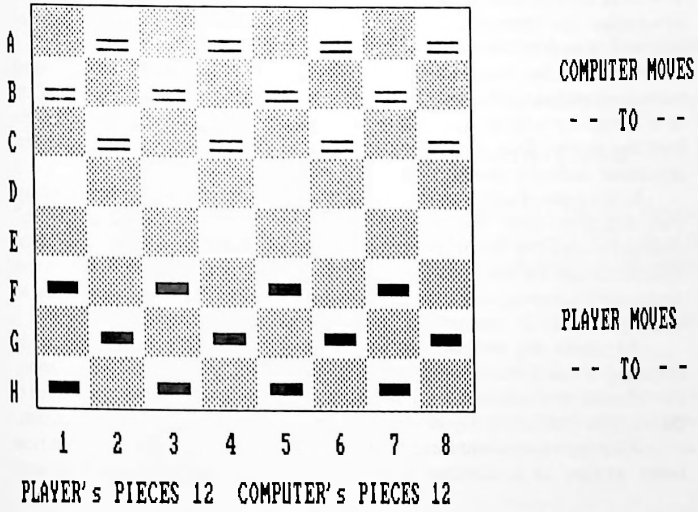
FIGURE 7.1. Graphical representation of checkers' pieces.

A piece is captured by jumping over it. Alternatively, a diagonal move of one square forward may be made. If a piece makes it to the opposite side of the board, it is crowned and displayed with a little cross in its middle. A crowned piece may move backward and forward.

If jumping (capturing) a piece is possible, then the jump must be made. Multiple jumps are also possible. In general, part of the strategy of checkers is to force your opponent to move to certain weak positions on the board by forcing him to capture your pieces. You, in turn, reply with a jump of your own but to a more strategic position or with more jumps (captures) than your opponent.

Moving a piece is accomplished by entering a from and to position. Each of the eight rows in the board is identified by the letters A to H, and the columns by the digits 1 to 8. A piece is thus identified by its position, such as F4 for the piece of row F and column 4. See Figure 7.2 for an illustration of the checkerboard as displayed by our program.

FIGURE 7.2. Checkers game example.



The checkers program is played as follows. First, the computer asks you to enter a P or C to indicate who is to make the first move. Press a P for you (the player) or a C for the computer. This is followed by the now-familiar "Press <SPACE> when ready" message. After pressing the space bar, the screen is cleared and the board is displayed.

Interacting with the computer is done by entering the from and to positions for the piece being moved. If you move first, you enter a move such as F4 E5 on the four dashes below the word PLAYER on the screen. This move is the piece on row F at column 4 being moved to row E at column 5. The computer checks that you have chosen a valid square to move from and to, plus whether or not you should be using another piece to make a jump. If there are any problems with your move, a beep will sound and you will have to enter a new move.

After your move is entered, it is highlighted on the screen by blinking the middle of the piece being moved and the middle of the square the piece is being moved to. This allows you to confirm that this is the move you intended. Confirm the move by pressing a Y for Yes or N for No: You *must* press either of these two keys. All other keystrokes are ignored. If Y is pressed, the move is made; if N is pressed, you are allowed to reenter a new move.

When it's the computer's turn to move, it will display its from and to move positions in the same manner as the moves you entered. This is followed by a message that you must press the space key to confirm the move. This is done to make you aware of what move the comptuer actually made.

In the case of continued jumps, where one jumped piece sets up the jump of another, the computer asks for you to enter the next "to" position where the "from" position continues from the previous move. This continues for each piece that can be jumped. In the case of the computer's jumping more than one of your pieces, it will continue to display a new "to" position, which you must acknowledge.

As pieces are jumped, the number of pieces left on the board is updated at the bottom of the screen. When either side shows a zero, the other player has won and the game is over. At the conclusion of a game, you are asked if you want to play again or end the program.

Although relatively long in code (about 270 lines), checkers has been written as a number of fairly discrete sections (subroutines)

which helps in understanding it. The overall flow of the program is as follows:

1. Determine who plays first, display the board, and initialize the board array and other variables. If the computer goes first, then go to step number 6.
2. Check to see if any jumps are possible by the player.
3. Get the player's "from" and "to" move positions, and validate them. If a jump is required and one has not been entered, beep and ask again for the move.
4. Blink the "from" and "to" squares on the board to let the player confirm the move.
5. If a piece is jumped, check to see if all twelve pieces have been captured for the end of the game. If a jump can continue, go back to step 3.
6. Create a list of all possible moves the computer can make, assigning a value to each. Sort the moves and randomly select one from the highest value of moves. Note that if the computer goes first, this step follows step 1.
7. Display the computer's move for the player to confirm; then make the move.
8. If a piece is jumped, check to see if the game has ended and whether a continued jump is possible.
9. Go to step 2 and continue with the next player's move.

Internal to the program, the board is represented by the two-dimensional 12 by 12 array B. The entries in B are integers in the range of −13 to +13, where the values are coded to represent the following:

- 0 A piece cannot be moved to this square. (It is used for border squares, and rows and columns 0, 1, 10, and 11.)
- 1 this square is empty. A piece can be moved to it.
- 2 to 13 this square contains one of the player's pieces, where its number is its index to the arrays RP and CP, for row/column of player.
- −2 to −13 this square contains a computer's piece, where the number indicates an index to arrays RC and CC.

This coding of the pieces allows a simple check for the placement of a checker. An array entry of 0 means that no move can be made to that square. An entry of less than −1 means that a square is occupied by the computer; if the entry is greater than 1, then it is occupied by the player. Last, an array entry of 1 means that the square is empty and a piece can be moved to it.

The indices to the four arrays RP, CP, RC, and CC mentioned above are used as follows. To evaluate which moves are possible by the computer and to also check if a jump is possible by any of the player's pieces, we could search through all entries in the board array, looking for each player's pieces. Instead, the RP and CP array lists where each of the player's and computer's pieces are in the B array (row and column). Thus, we need only look at the twelve entries in these arrays instead of the sixty-four possible positions in B.

Two other arrays are also used, TP and TC, for the type of a piece. A type code of 0 means it is a regular piece; a 1 indicates that the piece has been crowned; and a 2 means that the piece has been jumped. The code that indicates pieces have been jumped is used to prevent moving pieces that have been captured.

Moving a piece requires converting the code of the "from" and "to" position (the "external position") to its internal B array position. This is done by adding a 2 to both the piece's row and column values. The "from" and "to" position is also converted to a row and column for display of the appropriate piece on the screen. Two subroutines are provided: one to display the player's pieces and one to display the computer's pieces.

Selecting the computer's move is done as follows. Each of the computer's pieces is examined by searching the RC and CC arrays. For each piece currently in use (not captured), it is determined whether the piece can move forward or forward and backward. It is also determined if the piece can jump forward over a player's piece and if it can jump backward (if it is a crown). For each possible move, a value is determined and saved in the variable VM (for value of move). The value is first set for each of the four moves just listed, and is adjusted as a lower value if the position to which the piece moved would allow the player to jump it in return.

In summary, the value of the moves is as follows:

- 8 Crown captures a piece without being captured
- 7 Crown captures a piece but is captured by the player.

- 6 A regular piece captures a piece but is not captured.
- 5 A regular piece captures a piece but is captured by a player.
- 4 A crown is moved without being captured.
- 3 A crown is moved but is captured.
- 2 A regular piece is moved without being captured.
- 1 A regular piece is moved but is captured by the player.

Note how the top four moves are all jumps. This forces the computer to make a jump move if one is possible, in accordance with the rules of the game.

The index number to the piece that can be moved, the row and column it can be moved to, and the value of the move are saved in the arrays NP, MR, MC, and VM. The index number, saved in the NP array, is used to look up the piece's current position in the RC and CC arrays. They become the computer's "from" position when it makes its move. The values in MR and MC are the "to" position for this move.

After all pieces are examined, the moves in the above arrays are sorted in descending sequence based on their value as found in array VM. After sorting, the 0th element in the above arrays is the highest valued more for the computer. It may be tied with elements 1, 2, or 3 and so on. Thus, the program checks to see if other moves might be chosen and, from those with this same value, randomly selects a move. The move is displayed for the player to confirm; then the move is made on the screen.

This type of move selection is somewhat simplistic. For example, the possible replies that a player could make in return to a move other than a possible jump of the current piece being checked are not considered. Furthermore, more value can be assigned to those moves that take over centrally located squares versus the edge of the board. There are numerous additional criteria that could be added to the value of the computer's possible moves. On the other hand, the code is fairly simple and short, and examines all possible moves.

This cannot always be said for the human player: It's interesting to note how often a piece that might be jumped is overlooked. Although not considered very "intelligent," our checkers program does play a consistent and steady game. Further inspiration for it is left to the ambitious reader. Program 7.1 shows the complete listing for our checkers game.

```
0 '  CHECK   Play the game of checkers
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  Play checkers with the computer: either human or computer can go
50 '  first.  Game is over when one player loses all his pieces.
60 '
70 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
80 DEFINT A-Z : RANDOMIZE VAL(MID$(TIME$,4,2)+RIGHT$(TIME$,2))
90 '
100 ' Arrays used:
110 '
120 DIM B(11,11)  ' board
130 DIM RC(11),CC(11),TC(11) ' Computer's pieces row,column,type
140 DIM RP(11),CP(11),TP(11) ' Player's pieces row,column,type
150 DIM NP(24),VM(24),MR(24),MC(24) ' Potential moves: piece,value,row,col
160 '
170 PRINT "CHECK  Game of Checkers" : PRINT
180 PRINT "Who goes first?  Press P for player, C for computer: ";
190 GOSUB 2610 : P$=C$ : IF INSTR("PpCc",P$)=0 THEN BEEP : GOTO 190
200 PRINT P$; : PRINT "Okay.  Now please press <SPACE> to begin game.";
210 WHILE INKEY$<>" " : WEND
220 '
230 ' initialize and display the board
240 '
250 COLOR 0,7 : CLS  ' black on white background
260 FOR I=0 TO 7      ' each row
270 LOCATE I*2+6,3 : COLOR 0,7 : PRINT CHR$(65+I); ' row letters
280   FOR J=1 TO 2
290   LOCATE I*2+J+4,6
300      FOR K=0 TO 7 ' each column
310      IF (I MOD 2)=1 THEN CLR=(K MOD 2)*4 ELSE CLR=((K+1) MOD 2)*4
320      COLOR ,CLR : PRINT SPC(5); ' black or red square background
330      NEXT K
340   NEXT J
350 NEXT I
360 COLOR ,7 ' back to white background
370 FOR C=7 TO 42 STEP 5 : LOCATE 22,C : PRINT C\5; : NEXT 'column numbers
380 LOCATE 7,55 : PRINT"COMPUTER MOVES";
390 LOCATE  9,56: PRINT"- -  TO  - -";
400 LOCATE 17,55: PRINT"PLAYER MOVES";
410 LOCATE 19,56: PRINT"- -  TO  - -";
420 LOCATE 24,4 : PRINT"PLAYER's PIECES 12";
430 LOCATE  ,24 : PRINT"COMPUTER's PIECES 12";
440 '
450 ' initialize arrays and display pieces
460 '
470 K=-1 : NP1=12 : NP2=12 ' number of pieces
480 FOR I=2 TO 9 : FOR J=2 TO 9 : B(I,J)=1 : NEXT : NEXT
490 FOR I=0 TO 2 : FOR J=2 TO 8 STEP 2 : K=K+1
500 R=I+1 : C=J-(I MOD 2) : GOSUB 2380  ' display computer's pieces
510 B(R+1,C+1)=-K-2 : RC(K)=R+1 : CC(K)=C+1 : TC(K)=0 ' save position
520 R=I+6 : C=J-1+(I MOD 2) : GOSUB 2300 ' display player
530 B(R+1,C+1)=K+2 : RP(K)=R+1 : CP(K)=C+1  : TP(K)=0 ' save position
540 NEXT : NEXT
550 '
560 ' who's first?  decide and begin play
570 '
```

```
580 IF P$="P" OR P$="p" THEN 660 ' player?
590 CP=INT(RND+9) : K=SGN(.5-RND) : IF K=0 THEN 590
600 RF=RC(CP) : CF=CC(CP)   ' move piece 9 or 10
610 RT=RF+1   : CT=CF+K      ' up one row, left or right
620 GOTO 1380  ' computer makes first move
630 '
640 ' get player's move
650 '
660 COLOR 0,7
670 FOR PP=0 TO 11
680 IF TP(PP)<>2 THEN GOSUB 1980 : IF MJ THEN 700 ' any jumps possible?
690 NEXT
700 LOCATE 19,56 : PRINT "- -  TO  - -";
710 P=1 : GOSUB 2180 : IF P=0 THEN BEEP : GOTO 700 ' get FROM position
720 RF=R : CF=C : PP=B(R+1,C+1)-2
730 LOCATE 19,65 : PRINT "- -";
740 P=10 : GOSUB 2180 : IF P=0 THEN BEEP : GOTO 700 ' get TO position
750 RT=R : CT=C
760 IF MJ THEN IF ABS(RF-RT)<>2 OR ABS(CF-CT)<>2 THEN 800 ELSE 810 ' jump move
770 RJ=0 : CJ=0
780 IF RF-RT=1 AND ABS(CF-CT)=1 THEN 830    ' regular forward move
790 IF RT-RF=1 AND ABS(CF-CT)=1 AND TP(PP)=1 THEN 830 ' crown moves backwards
800 BEEP : GOTO 660
810 RJ=RF+(RT-RF)\2 : CJ=CF+(CT-CF)\2  ' ok to jump RJ,CJ
820 IF B(RJ+1,CJ+1)>-2 THEN 800        ' if a piece is there!
830 LOCATE 21,55,1 : PRINT "MOVE OKAY (Y/N)?";
840 '
850 '  highlight move for player to verify
860 '
870 COLOR 4,0 : C$="" : K=0
880 WHILE C$=""
890 K=(K+1) MOD 32 : C$=INKEY$ ' enter Y or N
900   LOCATE (RF-1)*2+6,(CF-1)*5+8,1
910   IF K<16 THEN PRINT " "; ELSE PRINT CHR$(223);
920   LOCATE (RT-1)*2+6,(CT-1)*5+8,1
930   IF K<16 THEN PRINT CHR$(223); ELSE PRINT " ";
940 WEND
950 COLOR 0,7 : LOCATE 21,55 : PRINT SPC(16);
960 IF C$="Y" OR C$="y" THEN 1050    ' move accepted
970 IF C$<>"N" AND C$<>"n" THEN 830  ' invalid entry
980 COLOR 4,0 ' reset highlight
990 LOCATE (RF-1)*2+6,(CF-1)*5+8,1 : PRINT CHR$(223);
1000 LOCATE (RT-1)*2+6,(CT-1)*5+8,1 : PRINT " ";
1010 GOTO 660 ' re-enter move
1020 '
1030 ' move player's piece
1040 '
1050 R=RF : C=CF : GOSUB 2470 : B(R+1,C+1)=1      ' erase FROM position
1060 RP(PP)=RT+1 : CP(PP)=CT+1 : IF RT=1 THEN TP(PP)=1  ' save new
1070 R=RT : C=CT : T=TP(PP) : GOSUB 2300 : B(R+1,C+1)=PP+2 ' display TO
1080 IF RJ=0 THEN 1190 ' move was not a jump
1090 R=RJ : C=CJ : GOSUB 2470 ' erase jumped square
1100 NP1=NP1-1 : GOSUB 2540 : IF NP1=0 THEN 1580 ' game over?
1110 CP=ABS(B(RJ+1,CJ+1))-2 : B(RJ+1,CJ+1)=1 : TC(CP)=2 ' clear computer piece
1120 GOSUB 1980 : IF NOT MJ THEN 1190 ' no continued jump
1130 R=RT : C=CT : COLOR 0,7  ' jump again
1140 LOCATE 19,56 : PRINT CHR$(64+R);" ";CHR$(48+C); ' set FROM
1150 GOTO 720  ' continue jump, get TO position
```

```
1160 '
1170 ' computer's turn, check possible moves for all active pieces
1180 '
1190 NM=-1
1200 FOR CP=0 TO 11
1210 CR=TC(CP) : R=RC(CP) : C=CC(CP) ' type,row,column of piece
1220 IF CR<2 THEN GOSUB 1700 ' check moves this piece
1230 NEXT
1240 '
1250 ' sort moves by value and pick one
1260 '
1270 FOR I=0 TO NM-1 : FOR J=I+1 TO NM
1280 IF VM(I)>=VM(J) THEN 1300
1290 SWAP VM(I),VM(J):SWAP NP(I),NP(J):SWAP MR(I),MR(J):SWAP MC(I),MC(J)
1300 NEXT : NEXT
1310 M=VM(0) : FOR I=0 TO NM : IF VM(I)<>M THEN LM=I : GOTO 1330
1320 NEXT : LM=NM+1
1330 PM=INT(RND*LM) : CP=NP(PM) ' pick piece to move
1340 RF=RC(CP) : CF=CC(CP) ' set move FROM
1350 RT=MR(PM) : CT=MC(PM) ' set move TO
1360 '
1370 ' display move selected by computer, then do it
1380 '
1390 COLOR 0,7
1400 LOCATE 9,56 : PRINT USING "& #";CHR$(63+RF);CF-1;
1410 LOCATE 9,65 : PRINT USING "& #";CHR$(63+RT);CT-1;
1420 LOCATE 11,55: PRINT "Press <SPACE> for move";
1430 WHILE INKEY$<>" " : WEND
1440 LOCATE 11,55: PRINT SPC(25);
1450 R=RF-1 : C=CF-1 : GOSUB 2470 : B(RF,CF)=1   ' erase the old position
1460 RC(CP)=RT : CC(CP)=CT : IF RT=9 THEN TC(CP)=1     ' save the new one
1470 R=RT-1 : C=CT-1 : T=TC(CP) : GOSUB 2380 : B(RT,CT)=-CP-2 ' and move to it
1480 IF ABS(RT-RF)<>2 THEN 660   ' computer didn't jump a piece
1490 RJ=RF+(RT-RF)\2 : CJ=CF+(CT-CF)\2   ' computer jumped RJ,CJ
1500 PP=ABS(B(RJ,CJ))-2 : B(RJ,CJ)=1 : TP(PP)=2  ' remove player's piece
1510 R=RJ-1 : C=CJ-1 : GOSUB 2470 ' erase jumped square
1520 NP2=NP2-1 : GOSUB 2540 : IF NP2=0 THEN 1580 ' game over?
1530 GOSUB 2080 : IF NOT MJ THEN 660 ' continue jumping?
1540 NM=-1 : GOSUB 1700 : GOTO 1270  ' find next jump for this piece
1550 '
1560 ' end of game
1570 '
1580 COLOR 0,7 : LOCATE 1,1 : PRINT SPC(79); : LOCATE 1,25
1590 IF NP1=0 THEN T$="Good going.  You won this time !!!"
1600 IF NP2=0 THEN T$="Sorry...  Computer wins this game."
1610 IF T$<>"" THEN PRINT T$;
1620 LOCATE 3,18 :  PRINT "Press Esc to exit or <SPACE> for another game.";
1630 GOSUB 2610
1640 IF ASC(C$)=27 THEN COLOR 7,0 : GOTO 9050 ' exit
1650 IF C$=" " THEN COLOR 7,0 : CLS : GOTO 180 ' new game
1660 BEEP : GOTO 1630
1670 '
1680 '  examine all moves for a piece
1690 '
1700 VM=2+CR*2
1710 MR=R+1 : MC=C+1 : GOSUB 1860 ' forwards
1720 MR=R+1 : MC=C-1 : GOSUB 1860
1730 IF CR=0 THEN 1760
```

```
1740 MR=R-1 : MC=C+1 : GOSUB 1860 ' backwards (crowns only)
1750 MR=R-1 : MC=C-1 : GOSUB 1860
1760 VM=6+CR*2
1770 MR=R+2 : MC=C+2 : GOSUB 1860 ' forward jumps
1780 MR=R+2 : MC=C-2 : GOSUB 1860
1790 IF CR=0 THEN 1820
1800 MR=R-2 : MC=C+2 : GOSUB 1860 ' backward jumps (crowns only)
1810 MR=R-2 : MC=C-2 : GOSUB 1860
1820 RETURN
1830 '
1840 '  check and save a possible computer move
1850 '
1860 IF B(MR,MC)<>1 THEN RETURN   ' square already occupied
1870 MJ=ABS(MR-R) : IF MJ=1 THEN 1890   ' not a jump move
1880 IF B(R+(MR-R)\2,C+(MC-C)\2)<2 THEN RETURN ' nothing to jump
1890 IF B(MR+1,MC+1)>1 OR B(MR+1,MC-1)>1 THEN VM=VM-1 : GOTO 1920 ' jump back?
1900 PP=B(MR-1,MC+1) : IF PP>1 THEN IF TP(PP-2)=1 THEN VM=VM-1 : GOTO 1920
1910 PP=B(MR-1,MC-1) : IF PP>1 THEN IF TP(PP-2)=1 THEN VM=VM-1
1920 IF NM>24 THEN RETURN ' examine at most 24 moves
1930 NM=NM+1 : NP(NM)=CP : VM(NM)=VM : MR(NM)=MR : MC(NM)=MC ' save move
1940 RETURN
1950 '
1960 '  can player at R,C jump a piece? MJ=0 no, MJ=-1 yes.
1970 '
1980 MJ=0 : R=RP(PP) : C=CP(PP)
1990 IF B(R-1,C+1)<-1 AND B(R-2,C+2)=1 THEN MJ=-1 : RETURN
2000 IF B(R-1,C-1)<-1 AND B(R-2,C-2)=1 THEN MJ=-1 : RETURN
2010 IF TP(PP)=0 THEN RETURN
2020 IF B(R+1,C+1)<-1 AND B(R+2,C+2)=1 THEN MJ=-1 : RETURN
2030 IF B(R+1,C-1)<-1 AND B(R+2,C-2)=1 THEN MJ=-1 : RETURN
2040 RETURN
2050 '
2060 '  can computer at R,C jump a piece? MJ=0 no, MJ=-1 yes.
2070 '
2080 MJ=0 : R=RC(CP) : C=CC(CP)
2090 IF B(R+1,C+1)>1 AND B(R+2,C+2)=1 THEN MJ=-1 : RETURN
2100 IF B(R+1,C-1)>1 AND B(R+2,C-2)=1 THEN MJ=-1 : RETURN
2110 IF TC(CP)=0 THEN RETURN
2120 IF B(R-1,C+1)>1 AND B(R-2,C+2)=1 THEN MJ=-1 : RETURN
2130 IF B(R-1,C-1)>1 AND B(R-2,C-2)=1 THEN MJ=-1 : RETURN
2140 RETURN
2150 '
2160 ' player enters a move
2170 '
2180 LOCATE 19,55+P,1 : GOSUB 2610 ' get row letter A-H
2190 IF ASC(C$)>96 THEN C$=CHR$(ASC(C$)-32)
2200 R=INSTR("ABCDEFGH",C$) : IF R=0 THEN BEEP : GOTO 2180
2210 LOCATE 19,55+P : PRINT C$;
2220 LOCATE 19,57+P,1 : GOSUB 2610 ' get column number 1-8
2230 C=INSTR("12345678",C$) : IF C=0 THEN BEEP : GOTO 2220
2240 LOCATE 19,57+P : PRINT C$;
2250 IF P=1 THEN IF B(R+1,C+1)>1 THEN RETURN ELSE P=0 : RETURN
2260 IF B(R+1,C+1)=1 THEN RETURN ELSE P=0 : RETURN
2270 '
2280 ' display player's piece
2290 '
2300 COLOR 4,0
2310 LOCATE (R-1)*2+6,(C-1)*5+7 : PRINT STRING$(3,223);
```

```
2320 IF T<>1 THEN RETURN
2330 LOCATE (R-1)*2+5,(C-1)*5+8 : PRINT CHR$(194); ' crown
2340 RETURN
2350 '
2360 ' display computer's piece
2370 '
2380 COLOR 4,0
2390 LOCATE (R-1)*2+6,(C-1)*5+7 : PRINT STRING$(3,205);
2400 IF T<>1 THEN RETURN
2410 LOCATE (R-1)*2+5,(C-1)*5+8 : PRINT CHR$(194); ' crown
2420 LOCATE (R-1)*2+6,(C-1)*5+8 : PRINT CHR$(207);
2430 RETURN
2440 '
2450 ' erase a square
2460 '
2470 COLOR 7,0
2480 LOCATE (R-1)*2+5,(C-1)*5+6 : PRINT SPC(5);
2490 LOCATE (R-1)*2+6,(C-1)*5+6 : PRINT SPC(5);
2500 RETURN
2510 '
2520 '  display number of pieces left
2530 '
2540 COLOR 0,7
2550 LOCATE 24,20 : PRINT USING "##";NP2;
2560 LOCATE 24,42 : PRINT USING "##";NP1;
2570 RETURN
2580 '
2590 ' get single character from keyboard
2600 '
2610 LOCATE ,,1 : C$="" : WHILE C$="" : C$=INKEY$ : WEND : RETURN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 7.1 CHECK - Checkers game

Backgammon

The game of backgammon has recently become quite popular. It has quite a heritage, having originated with the Arabs many thousands of years ago. Learning to play is somewhat difficult, but once understood, backgammon is a fun and fast-paced game, combining luck and skill.

The overall objective is to move your pieces (men) from one end of the board to the other in a race against your opponent. Luck enters

the game because the distance of your moves is determined by rolling the dice. The skill part is knowing which pieces to move and where to move them. Figure 7.3 illustrates the opening-board layout of backgammon and the directions that each player's stones are moved.

To adequately describe the rules and how to play would take many pages, so instead, we will provide only a cursory look at them. Many books have been written on learning the game—you should have no problem in finding one, if you're interested.

FIGURE 7.3. Opening backgammon board positions.

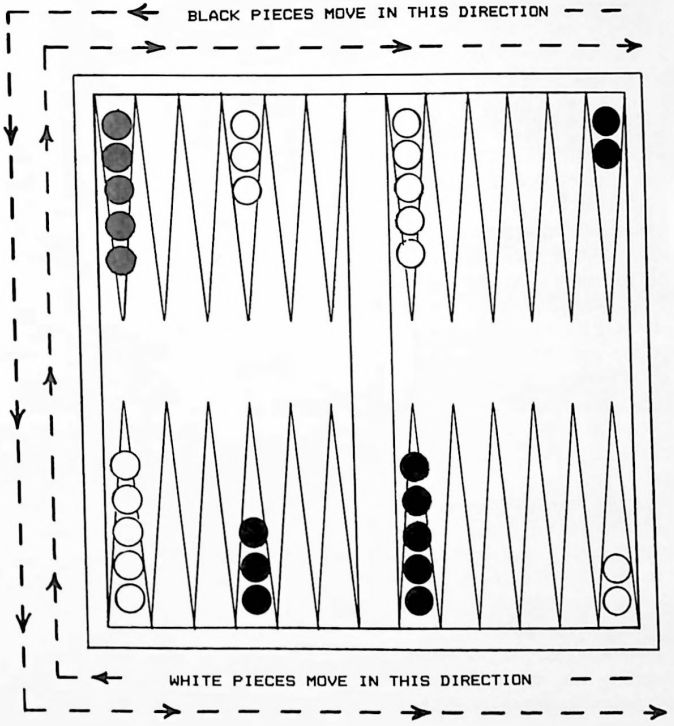


Figure 7.4 illustrates the backgammon program being played. In it, your pieces are white letter o's; your opponent's pieces are the dark squares. You must move your pieces in a clockwise manner from the lower-right corner row of the board to the top-right corner. Your opponent moves in the opposite direction. The right side of the board is usually called the "home board" or "inner table," and the left side is referred to as the "outer board" or "outer table."

Movement of men is from point to point according to the numbers on the dice. The following restrictions apply: You may not move your man to a point containing two or more of your opponent's men (which is known as a "block"). If a point is empty, you may move to it. Two men may be moved: one for each number on each die. Alternatively, one piece may be moved twice: once for the first die and one for the second die, as long as the man does not land on a blocked point after the move for the first die.

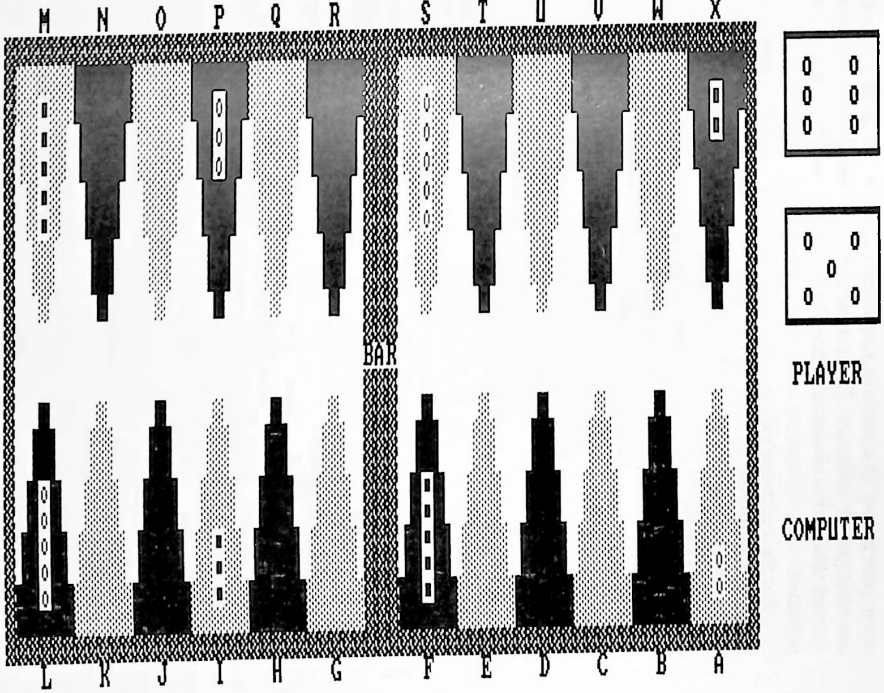
You may move a man onto a point containing only one of your opponent's pieces, which will cause the following to happen. The man located there is moved to the center line of the board (the "bar"). Once on the bar, that man must reenter the board on its inner table; that is, at the opposite end of the board from the direction in which that man was moving. Thus, that man must begin its travels around the board at the furthest point from the finish. Needless to say, being put on the bar can be very disruptive to your play.

Once all your men are moved to the inner table, you move them off the board (which is known as bearing off). You remove a man on the points for which a dice number is rolled. If no men are on the point designated by the number on the dice, a stone beyond that point is moved forward; conversely, if there are no men beyond that point, a piece closer in may be borne off. The first player to bear off all his pieces wins the game.

So much for a cursory review of the rules. Our program of backgammon is played as follows. When the program is begun, you enter a P or C to indicate who moves first, as in checkers. The screen is then cleared, and the board and men are displayed.

The board is arranged so that all points have a letter: from A to X. This arrangement was chosen since the letters are in the order of the player's movement about the board. To indicate the movement of a particular piece, you enter its point letter and one of the two dice numbers that was rolled, such as A2 for a man from point A moved two points forward—to point C.

FIGURE 7.4. Backgammon game program being played.



In general, interacting with the backgammon program is very similar to our checkers program. To roll the dice, you press the space key and the dice change randomly for a few moments. You then enter your move. The piece is moved and the points on the screen are changed accordingly. This procedure is then repeated for the second die. In the event that doubles are thrown, the process is repeated.

For the computer's move, you are also requested to press the space key to roll the dice. After the computer determines its move, the move is displayed for you to see. You press the space key again to confirm the move being made. The screen is then updated, as in the player's move.

If a piece is moved to the bar, then the move is requested in the form "BAR TO:," where you enter the point letter that corresponds to one of the numbers on the dice you have just rolled. For instance, a roll of 2 and 4 might be used in moving off the bar to points B or D. In the event that both points are blocked, the program will indicate that you cannot move. The computer moves off the bar in a similar manner.

The program for backgammon is the longest and most involved program in this book. Part of this is due to the rules of the game that must be embodied in the program, and also for a number of complex situations that can occur during play. The difficult situations are primarily when all points to which you might move are blocked, and thus no move can be made with either of the dice.

The game begins by first displaying the board on the screen. The board consists of a border, the vertical bar, the twenty-four points, and the dice. The format and program code for the dice are identical to that used in the craps gambling game in Chapter 6.

The array P$ contains ten character strings which make up a "printable" point. The points are printed from either top to bottom for the lower half of the board and bottom to top for the top half of the board. Note that the columns for the points to the right of the bar are offset by an additional three spaces to make room for the bar.

Displaying the men on a point is done as follows. A player's man is represented by the small letter "o," ASCII character code 111, and the computer's man is represented by a small square symbol, character code 254. Two subroutines are provided, which take a point number in the variable P as a parameter and display the men on it. (It first erases all men that are currently there and then displays

the new number of men.) This is done because the routine may be used to delete men from a point when they move to the bar or when their number increases as a new man is added to a point. Only seven men may be displayed on a point at a time. If it is necessary to place more than seven stones on a point, a number in the bottom-most man is displayed to indicate the additional number of men.

One other subroutine is also provided for displaying men on the bar. It takes as a parameter the variable BAR which, if negative, indicates that it must display that many of the computer's pieces on the bar; if positive, it must display that many of the player's pieces. If BAR equals zero, then the subroutine checks which side of the bar must be cleared by checking the variables PBAR or CBAR. This is required since it cannot be determined which player is no longer on the bar when the BAR variable is zero.

As in the checker program, a number of arrays are used for representing the board and for the computer to determine its move. The single-dimension array B is the board. It is indexed from 0 to 23 where 0 is the inner table of the computer and 23 is the innermost point of the player. Thus, the computer moves from 0 to 1, 2, 3, and so on, and the player moves from 23 to 22, 21, and so on.

A zero entry in B indicates that a point is empty; a positive value is the number of player's pieces on the point; and a negative value is the number of computer pieces. Thus, B(23) is initially a +2 for the player's two pieces that are furthest from the inner table, and B(0) is −2 for the computer's two pieces that are furthest from the inner table.

Three variables for both the computer and player control much of the play: They are POFF, PBAR, and PBEAR for the player; and COFF, CBAR, and CBEAR for the computer. These variables are the total number of pieces that have yet to be borne off, the number of pieces on the bar, and how many pieces are in the inner table, respectively.

POFF and COFF range from 0 to 15. Once at 15, the game is over: All pieces have been borne off. PBAR and CBAR are 0 when no pieces are on the bar. PBEAR and CBEAR range from 0 to 10; they count the number of pieces moved into the inner table. When either of these variables becomes 10, it means that a player may begin to bear off his pieces.

Note that while bearing off, your opponent might be on the bar. He would then reenter on your inner table, possibly putting *you* on

the bar. PBEAR would then be decremented from 10 to 9. You would then have to move that piece completely around the board before bearing off any more pieces.

Before the player makes his move, the program checks to see if a move is possible with either of the dice. If neither die may be used (an unlikely event but nevertheless possible), then a message to that effect is printed. If one or both dice may be used, then the player's move is requested.

First, the point letter is requested. It is edited to ensure that it's an alphabetic letter from A to X and that the point has at least one piece located on it. The die number is then requested. It must be equal to one of the two dice that has just been rolled. The point that the man is moving to is then checked. You may move to any point that is not occupied by two stones of the opponent. If one of your men is on the bar (PBAR is greater than 0), then you are asked for a point to bear off to, as mentioned earlier. The point moved to is also validated in a manner similar to what was described above.

After the player's move is made, the computer then chooses a move. A number of arrays are used which contain the possible moves that the computer can take. They will be described as the procedures for determining the computer's move are discussed. (It is similar to the techniques used in the checkers program.)

First, the computer checks if any of its men are on the bar; if so, the move off is handled separately from a routine move. In sequence, all points are examined for each of the computer's men. This is done by looking at all negative elements in the B array. A positive-valued B array entry means that a point holds a player's man. A zero value means that the point is empty.

It is then determined if a man from this point can be moved with the first die. The point being moved to must either contain the computer's pieces, be empty, or have at most one player's piece. This is done by simply checking if that point's position in the B array is less than two.

If a move is possible, it is saved as a potential move with a value equal to its distance from the inner table plus an additional weighting factor. A 50 is added to the value of a move if it creates a block; 25 is added if it puts a player's piece on the bar. Both 25 and 50 might be added if a block is created at the same time the player is put on the bar.

The arrays VM and MP are used for saving all possible moves. It is set up so that MP(0,i) are the point numbers of the possible moves

from a point with the first die, and MP(1,j) are the moves possible from that point with the second die. After all moves for a die have been found, they are sorted by the values in VM. The VM array contains the value of the moves that have been found for this die as saved in array MP. The process is done for each die.

After all possible moves for each die have been found and put in sequence based on their value, one is selected for each die. This is usually MP(0,0) for die 1 and MP(1,0) for die 2. The exception is when the program attempts to move a single man from one point using the combined values of the dice. This may happen since the program examines moves for each die separately. Thus, if MP(0,0) equals MP(1,0) and there is only one man at that point, then another move for one of the dice is used. After a move is selected, it is shown to the player to acknowledge; then the move is made on the screen.

The overall basis of judgment being used for the computer's moves is "advance the pieces furthest back over those that may be closer to the inner table." The move value is also adjusted for those moves that put a player's man on the bar and those moves that create a blocked point. The best move is one in which the furthest piece out is advanced so that it puts the opponent on the bar.

Note the similarity of this program with the checkers program. The backgammon procedure does no in-depth evaluation of possible player rolls of the dice and piece movement, but once again, there is at least a degree of thoroughness because all points are examined.

When its men are on the bar, the computer looks for its moves as follows. First it selects the die which, if used, might create a block. If neither die will create a block, then the die whose move would put a player's man on the bar is chosen. If this check fails, the computer then picks either the first die or the second, depending on which point can be reached. Occasionally, a move off the bar is not possible. If this happens, a message to that effect is printed and the computer must forfeit its turn.

The listing for our backgammon game is in Program 7.2.

```
0 '  BACG   The game of BACKGAMMON
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  Play backgammon with the computer.  Either human or computer
50 '  may go first.  Player who bears off all pieces first wins game.
60 '
70 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS : DEFINT A-Z
80 RANDOMIZE VAL(MID$(TIME$,4,2)+RIGHT$(TIME$,2))
90 DIM B(23),P$(8),VM(99),MP(1,99)
```

```
100 PRINT "BACG  Bacgammon Game" : PRINT
110 PRINT "Who goes first?  Press P for player, C for computer: ";
120 GOSUB 3140 : P$=C$ : IF INSTR("PpCc",P$)=0 THEN BEEP : GOTO 190
130 PRINT P$; : PRINT "Okay.  Now please press <SPACE> to begin game.";
140 WHILE INKEY$<>" " : WEND
150 '
160 ' P$() contains "point" image for board
170 '
180 P$(0)=STRING$(5,219):P$(1)=P$(0):P$(2)=CHR$(222)+STRING$(3,219)+CHR$(221)
190 P$(3)=P$(2):P$(4)=" "+STRING$(3,219)+" ":P$(5)=P$(4)
200 P$(6)=" "+CHR$(222)+CHR$(219)+CHR$(221):P$(7)=P$(6):P$(8)="  "+CHR$(219)
210 '
220 ' display board
230 '
240 COLOR 5,0 : CLS
250 LOCATE 2,1:PRINT STRING$(65,178); ' top
260 FOR R=3 TO 23
270 LOCATE R,1:PRINT CHR$(178);       ' left side
280 LOCATE R,32:PRINT STRING$(3,178); ' bar in middle
290 LOCATE R,65:PRINT CHR$(178);      ' right side
300 NEXT
310 LOCATE 24,1:PRINT STRING$(65,178); ' bottom
320 LOCATE 13,32 : COLOR 7,5 : PRINT "BAR";
330 '
340 ' display points
350 '
360 CLR=12: L=0 : COLOR CLR,0
370 FOR C=2 TO 60 STEP 5
380 IF C=32 THEN C=35 ' leave room for bar
390 COLOR CLR : FOR I=0 TO 8:LOCATE 3+I,C:PRINT P$(I);:NEXT ' point down
400 LOCATE 1,C+2:PRINT CHR$(L+77);
410 IF CLR=12 THEN CLR=7 ELSE CLR=12
420 COLOR CLR : FOR I=0 TO 8:LOCATE 23-I,C:PRINT P$(I);:NEXT ' point up
430 LOCATE 25,C+2:PRINT CHR$(76-L);
440 L=L+1 : NEXT
450 COLOR 7,0:R=2:C=68:GOSUB 2720:R=8:GOSUB 2720 ' dice boxes
460 COLOR 7,0:R=3:C=70:GOSUB 2810:R=9:GOSUB 2810 ' dice numbers
470 LOCATE 14,69:PRINT"PLAYER";    : LOCATE 20,68:PRINT"COMPUTER"
480 '
490 ' initialize variables and display pieces
500 '
510 FOR I=0 TO 23 : B(I)=0 : NEXT
520 CBAR=0 : COFF=0 : CBEAR=0 : PBAR=0 : POFF=0 : PBEAR=0
530 P=11 : B(P)=-5 : GOSUB 2440 : P=12 : B(P)=+5 : GOSUB 2440
540 P=8  : B(P)=+3 : GOSUB 2440 : P=15 : B(P)=-3 : GOSUB 2440
550 P=5  : B(P)=+5 : GOSUB 2440 : P=18 : B(P)=-5 : GOSUB 2440
560 P=0  : B(P)=-2 : GOSUB 2440 : P=23 : B(P)=+2 : GOSUB 2440
570 '
580 ' determine who's first and begin play
590 '
600 IF P$="C" OR P$="c" THEN 1300
610 '
620 ' roll dice and check if a move is possible
630 '
640 COLOR 7,0 : GOSUB 2920 ' roll dice
650 SD1=-1 : SD2=-1      ' dice move status
660 IF PBAR>0 THEN 1060 ' must move off bar
670 FOR I=0 TO 23
680 IF B(I)<1 THEN 770  ' player on this point?
```

```
690 IF D1=0 THEN 730     ' die 1 already used?
700 IF I-D1<0 THEN IF PBEAR=10 THEN 720 ELSE 730   ' bear off with die 2?
710 IF B(I-D1)<-1 THEN 730        ' check move to point
720 SD1=0 : IF NOT SD2 THEN 780  ' ok to move with die 1
730 IF D2=0 THEN 770      ' die 2 already used?
740 IF I-D2<0 THEN IF PBEAR=10 THEN 760 ELSE 770   ' bear off with die 1?
750 IF B(I-D2)<-1 THEN 770        ' check move to point
760 SD2=0 : IF NOT SD1 THEN 780  ' ok to move with die 2
770 NEXT
780 IF SD1 OR SD2 THEN GOSUB 3050 ' can't move a die message
790 IF SD1 AND SD2 THEN 1300       ' no moves possible
800 COLOR 7,0 : LOCATE 16,68 : PRINT SPC(12);
810 '
820 ' player makes a regular move
830 '
840 GOSUB 2290 ' get move
850 IF B(PT)<=0 THEN BEEP : GOTO 840      ' computer's point
860 IF PT-D<0 THEN IF PBEAR=10 THEN 900 ELSE BEEP : GOTO 840 ' bear off?
870 IF B(PT-D)<-1 THEN BEEP : GOTO 840     ' cannot move here
880 IF D1=D AND SD1 THEN BEEP : GOTO 840  ' can't use die 1
890 IF D2=D AND SD2 THEN BEEP : GOTO 840  ' can't use die 2
900 P=PT    : B(P)=B(P)-1 : GOSUB 2440      ' erase from point
910 P=PT-D : IF P>-1 THEN 930                ' bearing off?
920 POFF=POFF+1 : IF POFF=15 THEN 2180 ELSE 990 ' up bear off count
930 IF B(P)=-1 THEN GOSUB 2050               ' put computer on bar
940 B(P)=B(P)+1 : GOSUB 2440                 ' move to new point
950 IF PT>5 AND P<6 THEN PBEAR=PBEAR+1     ' up inner table count
960 '
970 ' process 2nd die and doubles if any
980 '
990 IF D1>0 AND D=D1 THEN D1=0 ELSE IF D2>0 AND D=D2 THEN D2=0
1000 IF D1<>0 OR D2<>0 THEN 660 ' one more die to move
1010 IF DBLS<>0 THEN D1=DBLS : D2=DBLS : DBLS=0 : GOTO 650 ' doubles
1020 LOCATE 16,68 : PRINT SPC(12); : GOTO 1300 ' computer's turn
1030 '
1040 ' player moves off bar
1050 '
1060 IF D1>0 THEN IF B(24-D1)>-2 THEN SD1=0
1070 IF D2>0 THEN IF B(24-D2)>-2 THEN SD2=0
1080 IF (SD1 AND SD2) OR (SD1 AND D2=0) OR (D1=0 AND SD2) THEN GOSUB 3050 : GOTO
 1300
1090 LOCATE 16,68 : PRINT "BAR TO - ";
1100 LOCATE 16,75 : GOSUB 3140
1110 IF C$>"Z" THEN C$=CHR$(ASC(C$)-32)
1120 D=ASC(C$)-64
1130 IF (D1>0 AND D1=D AND NOT SD1) THEN 1160 ' ok to use die 1?
1140 IF (D2>0 AND D2=D AND NOT SD2) THEN 1160 ' or die 2?
1150 BEEP : GOTO 1090
1160 PBAR=PBAR-1 : BAR=PBAR : GOSUB 2530 ' clear bar
1170 P=24-D : IF B(P)=-1 THEN GOSUB 2050 ' put computer on bar?
1180 B(P)=B(P)+1 : GOSUB 2440 ' display piece
1190 GOTO 990
1200 '
1210 ' move player's piece to BAR
1220 '
1230 PBAR=PBAR+1 : BAR=PBAR : GOSUB 2530
1240 IF P<6 THEN PBEAR=PBEAR-1
1250 B(P)=0 : GOSUB 2440
1260 RETURN
```

```
1270 '
1280 ' computer rolls dice and moves
1290 '
1300 COLOR 7,0 : GOSUB 2920
1310 SD1=-1 : SD2=-1
1320 IF CBAR>0 THEN 1870 ' on BAR ?
1330 NM=-1 : K=0 : D=D1 : GOSUB 1620 : NM1=NM ' find all moves for 1st die
1340 NM=-1 : K=1 : D=D2 : GOSUB 1620 : NM2=NM ' and moves for 2nd die
1350 IF NM1>-1 AND NM2<0 THEN D2=0 : SD1=0 : B1=0 : GOTO 1420
1360 IF NM2>-1 AND NM1<0 THEN D1=0 : SD2=0 : B2=0 : GOTO 1420
1370 FOR B1=0 TO NM1 : FOR B2=0 TO NM2
1380 IF MP(0,B1)<>MP(1,B2) THEN 1440 ' select moves
1390 IF CBEAR=10 AND COFF=14 THEN 1440 ' only one piece left
1400 IF MP(0,B1)=MP(1,B2) AND B(MP(1,B2))<-1 THEN 1440 ' same point?
1410 NEXT : NEXT
1420 IF (SD1 AND D1>0) OR (SD2 AND D2>0) THEN GOSUB 3050 ' can't move a die
1430 IF SD1 AND SD2 THEN 640
1440 IF D1>0 THEN D=D1 : P=MP(0,B1) : D1=0 : GOSUB 1520 : IF COFF=15 THEN 2180
1450 IF D2>0 THEN D=D2 : P=MP(1,B2) : D2=0 : GOSUB 1520 : IF COFF=15 THEN 2180
1460 IF D1=0 AND D2=0 THEN IF DBLS=0 THEN 640 ' check for doubles to move again
1470 IF D1=0 AND D2=0 AND DBLS THEN D1=DBLS : D2=DBLS : DBLS=0
1480 GOTO 1320
1490 '
1500 ' move computer's piece
1510 '
1520 LOCATE 22,68 : PRINT USING "MOVE & #";CHR$(88-P);D: : GOSUB 2120
1530 B(P)=B(P)+1 : GOSUB 2440 ' clear old piece
 540 P=P+D : IF P>23 THEN COFF=COFF+1 : RETURN
 550 IF B(P)=1 THEN GOSUB 1230 ' put player on bar?
 560 B(P)=B(P)-1 : GOSUB 2440  ' display piece
 570 IF P-D<18 AND P>17 THEN CBEAR=CBEAR+1
 580 RETURN
1590 '
1600 ' check computer's potential moves
1610 '
1620 IF D=0 THEN RETURN
1630 FOR P=0 TO 23
1640 IF B(P)>-1 THEN 1750 ' piece on this point?
1650 IF CBEAR<>10 THEN 1680  ' check if bearing
1660 IF P+D=24 THEN VM=10 ELSE VM=23-P ' bear off value
1670 GOTO 1730
1680 IF P+D>23 THEN 1750  ' can move be made?
1690 IF B(P+D)>1 THEN 1750 ' player has a blot
1700 VM=23-P  ' advance piece
1710 IF B(P+D)=+1 THEN VM=VM+25  ' put player on bar
1720 IF B(P+D)=-1 AND CBEAR<>10 THEN VM=VM+50  ' build a blot
1730 NM=NM+1 : IF NM>99 THEN 1760 ' check at most 99 moves
1740 VM(NM)=VM : MP(K,NM)=P
1750 NEXT
1760 IF NM<0 THEN RETURN
1770 '
1780 ' sort moves by value and return
1790 '
1800 FOR I=0 TO NM-1 : FOR J=I+1 TO NM
1810 IF VM(I)<VM(J) THEN SWAP VM(I),VM(J):SWAP MP(K,I),MP(K,J)
1820 NEXT : NEXT
1830 RETURN
1840 '
```

```
1850 ' find a move for computer off BAR
1860 '
1870 B1=2 : IF D1>0 THEN B1=B(D1-1) ' use die 1?
1880 B2=2 : IF D2>0 THEN B2=B(D2-1) '  or 2?
1890 IF B1=1 OR B1=-1 THEN P=D1-1 : D1=D2 : D2=0 : SD1=0 : GOTO 1970
1900 IF B2=1 OR B2=-1 THEN P=D2-1 : D2=0 : SD2=0 : GOTO 1970
1910 IF B1<1 THEN P=D1-1 : D1=D2 : D2=0 : SD1=0 : GOTO 1970
1920 IF B2<1 THEN P=D2-1 : D2=0 : SD2=0 : GOTO 1970
1930 GOSUB 3050 : GOTO 640
1940 '
1950 ' move computer off BAR
1960 '
1970 LOCATE 22,68 : PRINT "BAR to ";CHR$(88-P); : GOSUB 2120
1980 CBAR=CBAR-1 : BAR=-CBAR : GOSUB 2530
1990 IF B(P)=1 THEN GOSUB 1230 ' put player on bar?
2000 B(P)=B(P)-1 : GOSUB 2440
2010 GOTO 1460
2020 '
2030 ' move computer's piece to BAR
2040 '
2050 CBAR=CBAR+1 : BAR=-CBAR : GOSUB 2530
2060 IF P>17 THEN CBEAR=CBEAR-1
2070 B(P)=0  : GOSUB 2440
2080 RETURN
2090 '
2100 ' acknowledge computer's move
2110 '
2120 LOCATE 23,67 : PRINT "Press <SPACE>"; : WHILE INKEY$<>" ": WEND
2130 LOCATE 22,68 : PRINT SPC(12); : LOCATE 23,67 : PRINT SPC(13);
2140 RETURN
2150 '
2160 ' end of game
2170 '
2180 IF COFF=15 THEN T$="Sorry... Computer won this time."
2190 IF POFF=15 THEN T$="Good going!!!  You won the game."
2200 LOCATE 12,20 : PRINT T$;
2210 LOCATE 13,14 : PRINT "Press Esc to exit or <SPACE> for another game.";
2220 GOSUB 3140
2230 IF ASC(C$)=27 THEN COLOR 7,0 : GOTO 9050 ' exit
2240 IF C$=" " THEN COLOR 7,0 : CLS : GOTO 110 ' new game
2250 BEEP : GOTO 2220
2260 '
2270 ' get player's move: point letter and die number
2280 '
2290 COLOR 7,0 : LOCATE 16,68 : PRINT "MOVE - -";
2300 LOCATE 16,73 : PRINT "- -";
2310 LOCATE 16,73,1 : GOSUB 3140
2320 C=ASC(C$) : IF C>96 THEN C=C-32 : C$=CHR$(C)
2330 IF C<65 OR C>88 THEN BEEP : GOTO 2300  ' point letter?
2340 PT=88-C : LOCATE 16,73 : PRINT CHR$(C);
2350 LOCATE 16,75 : PRINT "-";
2360 LOCATE 16,75,1 : GOSUB 3140 : IF C$=CHR$(13) THEN 2300
2370 C=ASC(C$) : IF C<49 OR C>54 THEN BEEP : GOTO 2350 ' from 1 to 6?
2380 D=C-48 : IF D<>D1 AND D<>D2 THEN BEEP : GOTO 2350 ' die number ok?
2390 LOCATE 16,75 : PRINT CHR$(C)
2400 RETURN
2410 '
2420 ' display pieces at board position B(P)
```

```
2430 '
2440 R=22+(P<12)*18 : K=11-P : IF P>11 THEN K=P-12
2450 C=K*5+4-(K>5)*3 : DIR=1+(P>11)*2
2460 IF P<12 THEN CLR=(K MOD 2)*-5+12 ELSE CLR=(K MOD 2)*5+7
2470 RR=R : CC=219 : NB=7 : COLOR CLR : GOSUB 2670
2480 R=RR : NB=ABS(B(P)) : CC=111-(B(P)<0)*143 : COLOR 7,0 : GOSUB 2640
2490 RETURN
2500 '
2510 ' display pieces for BAR: <0 computer, >0 player
2520 '
2530 COLOR 5,0 : IF BAR<>0 THEN 2570
2540 IF PBAR=0 THEN LOCATE 15,33 : PRINT CHR$(178);
2550 IF CBAR=0 THEN LOCATE 11,33 : PRINT CHR$(178);
2560 RETURN
2570 C=33 : R=11-(BAR>0)*4 : DIR=1+(BAR<0)*2
2580 RR=R : CC=178 : NB=7 : GOSUB 2670
2590 R=RR : NB=ABS(BAR) : CC=111-(BAR<0)*143 : COLOR 7,5 : GOSUB 2640
2600 RETURN
2610 '
2620 ' print pieces on screen at row,column R,C
2630 '
2640 IF NB<7 THEN 2670
2650 LOCATE R,C : PRINT USING"#";NB-6
2660 NB=6 : R=R+DIR
2670 FOR I=0 TO NB-1
2680 LOCATE R,C : PRINT CHR$(CC); : R=R+DIR
2690 NEXT
2700 RETURN
2710 '
2720 ' draw dice box
2730 '
2740 LOCATE R,C : PRINT CHR$(218);STRING$(7,196);CHR$(191);
2750 FOR I=1 TO 3:LOCATE R+I,C:PRINT CHR$(179);SPACE$(7);CHR$(179); : NEXT
2760 LOCATE R+4,C : PRINT CHR$(192);STRING$(7,196);CHR$(217);
2770 RETURN
2780 '
2790 ' display one die
2800 '
2810 FOR I=0 TO 2:LOCATE R+I,C:PRINT"       ";:NEXT
2820 D=INT(RND*6)+1 : ON D GOTO 2830,2840,2850,2860,2870,2880
2830 LOCATE R+1,C+2:PRINT"o";:RETURN
2840 LOCATE R,C:PRINT"o";:LOCATE R+2,C+4:PRINT"o";:RETURN
2850 LOCATE R,C:PRINT"o";:LOCATE R+1,C+2:PRINT"o";:LOCATE R+2,C+4:PRINT"o";:RETU
RN
2860 LOCATE R,C:PRINT"o   o";:LOCATE R+2,C:PRINT"o   o";:RETURN
2870 LOCATE R,C:PRINT"o   o";:LOCATE R+1,C+2:PRINT"o";:LOCATE R+2,C:PRINT"o   o"
;:RETURN
2880 LOCATE R,C:PRINT"o   o";:LOCATE R+1,C:PRINT"o   o";:LOCATE R+2,C:PRINT"o
o";:RETURN
2890 '
2900 ' roll the dice
2910 '
2920 LOCATE 7,69 : PRINT SPC(7);
2930 LOCATE 17,67 : PRINT "Press <SPACE>";
2940 LOCATE 18,67 : PRINT "to roll  dice";
2950 WHILE INKEY$<>" " : WEND
2960 LOCATE 17,67 : PRINT SPC(13); : LOCATE 18,67 : PRINT SPC(13);
2970 FOR K=1 TO 3
```

```
2980   R=3:C=70:GOSUB 2810:D1=D : R=9:GOSUB 2810:D2=D : SOUND 1999,.25
2990 NEXT
3000 DBLS=0 : IF D1=D2 THEN DBLS=D1 : LOCATE 7,69 : PRINT "DOUBLES";
3010 RETURN
3020 '
3030 ' can't move message
3040 '
3050 LOCATE 17,68 : PRINT "Can't move"; : LOCATE 18,70
3060 IF SD1 AND D1>0 THEN PRINT D1;
3070 IF SD2 AND D2>0 THEN PRINT D2;
3080 LOCATE 19,67: PRINT "Press <SPACE>"; : WHILE INKEY$<>" ": WEND
3090 LOCATE 19,67 : PRINT SPC(13);
3100 RETURN
3110 '
3120 ' get single character from keyboard
3130 '
3140 LOCATE ,,1 : C$="" : WHILE C$="" : C$=INKEY$ : WEND : RETURN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 7.2 BACG - Backgammon game

Variations

Here are a few possible variations in the board games that can be considered.

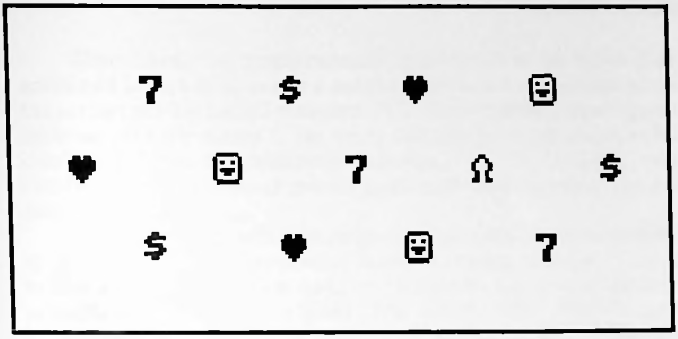
1. Provide an option for the checkers game to play with all crown pieces.

2. Modify the computers procedures in checkers for valuing its potential moves. Put more weight on moving or capturing crowns, for instance.

3. Convert the two games to a graphics mode. Each game's pieces could be coded for a DRAW or PUT statement that would be used to display them. See Chapters 8 and 9 for more details on using these graphics statements.

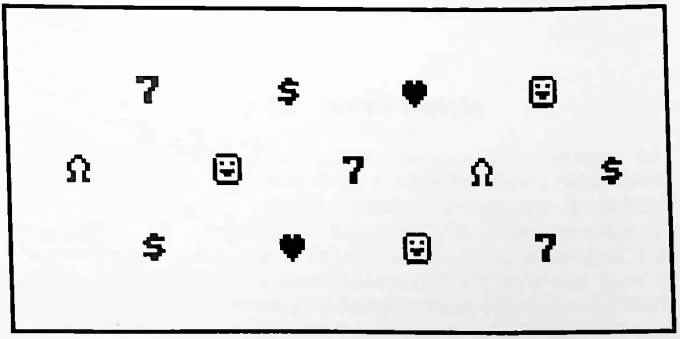
4. Modify the backgammon game's procedures for valuing its potential moves. Examine moves where one man is moved the total number shown on the dice as opposed to moving each man separately.

5. Allow wagers to be made prior to playing backgammon. Add a doubling-cube option. This is a gambling ploy where the stakes for the game double, quadruple, and so on when a player thinks he has a formidable advantage.



8

Action Games



Mention computer games to someone and probably the first thing that will come to his mind is an image of a rocket flying across a screen or an animated figure chasing or being chased by something. Action games test the player's reflexes, timing, and endurance: They involve a careful mixture of appealing images, an easy system of game control, increasing levels of playing difficulty, and a simple goal to achieve. Designing a good animated game requires a fair amount of programming skill plus a sense of choreography and artistry.

This chapter develops six animated action games. They range from a simple bouncing ball that is batted around the screen to an imitation of the Atari arcade game of "Asteroids." For the PC owner, four of the games will work on either type of adapter, that is, both the monochrome and color/graphics adapters, and two are designed so that they work only on the color/graphics adapter. All games will run on the PCjr.

CODING ANIMATED IMAGES

An animated game requires moving one or more images about the screen. Two methods can be used in coding the images for the games. A character mode approach can be taken in which the image is designed using groups of the PC's character codes. They are then color-coded with a COLOR statement, positioned on the screen with a LOCATE statement, and then displayed using the PRINT statement. Some programs POKE the character codes directly into the display adapter's memory buffer. In either case, a number of statements must be processed each time the image is displayed.

Alternatively, a graphics mode approach can be taken. The animated image is coded as a series of points that are displayed on the screen via the BASIC statement PUT. This statement takes a point address and the name of an array that contains the points to be displayed. Thus, the potentially numerous COLOR, LOCATE, and PRINT statements in the character mode approach are replaced by a single statement.

Another benefit with animation in the graphics mode is the form of the image itself. You are no longer limited to selecting from a limited character set when designing the image being used. Instead, you may use the graphics mode LINE, DRAW, PSET, PRESET, and PAINT statements to create highly detailed figures where the only limitation is your imagination and the resolution of the screen mode being used.

Of course, there are always some tradeoffs to be considered. In the character mode, you have sixteen colors to work with, whereas graphics on the PC is limited to four (or just black and white for high-resolution graphics). For PCjr owners, there are additional graphics modes available in its Cartridge BASIC. These include an eight-color medium-resolution mode, a four-color high-resolution mode, and a sixteen-color low-resolution mode. Unfortunately, these modes are not available on the PC.

The character mode image design process need not be a total washout. A clever choice of the PC's extended characters can often create some more-than-acceptable images for animation. And finally, a character-mode animated game will have the broadest base of potential users: All PC's and PC compatibles can function with character-mode graphics.

Moving Animated Images on the Screen

A fundamental aspect to animated games is controlling image movement. As an image changes position on the screen, a number of things must be considered. Most important is the erasure of the image from its previous position. This can be done in two ways: First, the original image may contain a certain amount of blank space about it which overlaps any remains of the image when it's moved.

Consider a 3-row by 3-column character-mode image. An additional row and column of blank spaces could be added to the top,

bottom, left, and right side of the image, increasing its size to 5 rows by 5 columns. As the image is moved one row or one column in any direction, the opposite row or column of blanks at the image's edge erases that portion of the image in the row or column that it left behind. This also means that movement is limited to a change in only one row or one column at a time. This is generally not that restrictive but can be if rapid image movement is achieved by moving the image two or more rows or columns at one time.

These considerations also apply to a graphics mode approach. An image as a set of 24 by 24 points is equivalent to the 3- by 3-character mode image just discussed. The edges about the image could include another eight background colored points in each direction, increasing the image's overall size to 40 by 40 points. Moving the image is then limited to changing its position by no more than eight points in any direction.

An alternative to this partial-erasure approach is to simply code another image that is blank and use it for a complete erasure of the original image each time the image is moved. The old image is overlayed by the blank one, which erases it. The image is then displayed again at some new position. The process is repeated as the image moves about the screen. This technique allows the original image to be moved anywhere on the screen instead of being limited to one row or one column at a time. For the 3- by 3-character mode image, this would mean that a second 3 by 3 set of blank characters would be maintained. As the original image is moved, it is erased from its previous position by the blank image.

For graphics-mode animation, the above approach can be used but an alternative exists that is somewhat simpler. To display a graphics-mode image, use the PUT statement. This statement requires a point position and the name of an array that contains the image's points, such as:

```
PUT (XPOINT,YPOINT), ROCKET
```

which places the graphics image contained in the array ROCKET at the point position XPOINT, YPOINT on the screen.

There is also an "action code" that determines the manner in which the points are placed on the screen. For example, the action code PSET places the image as is at the position defined. Another

code, PRESET, inverts the points and reverses their color before placing them on the screen. The action code is placed after the array name in the statement:

```
PUT (XPOINT,YPOINT), ROCKET, PSET
```

Of particular use for animation is the XOR action code. This code places the array's points on the screen but in a special manner. It resets the screen's points back to their original values if the screen already contains a copy of the image being PUT at that position. The first PUT causes the image to appear and the second PUT erases the image. Actually, the screen is restored to its original state, which is different from the manner in which erasure was considered in character-mode statements. The XOR action code is the default state for the PUT. If no code is defined, then it assumes XOR.

The significance of erasing an image versus restoring the screen to its original form depends on the background of the screen that is being used. If there are any background images that must be preserved, then the graphics mode XOR vastly simplifies the problems of image movement. Consider a background of planets and stars for a space game. As a rocket moves about, these images need to be maintained; otherwise, they would be slowly erased as the game is played. The XOR option takes care of this.

Using character-mode graphics to restore background imagery is somewhat complicated. There is no XOR equivalent for the COLOR, LOCATE, and PRINT statements used in character-mode image movement. Instead, some type of processing using the SCREEN function would be required to examine the screen prior to moving an image, so that portion of the screen could be restored after the image is moved.

Many animated games have no background problems because of the manner in which the game is designed. Not including any special background imagery is one solution—the overall color of the background is often more than sufficient. Another approach is to control image movement so that a player cannot move on top of background figures. This is similar to maze games where the object being moved is limited to pathways in the maze. The backgrounds in the character-mode games that follow are simple enough so that restoring them is not necessary.

Player Control of Animated Image Movement

Interacting with an animated game is generally kept as simple as possible. The typical form of interaction is controlling the direction in which one of the images moves. This can be via cursor control keys, a joystick, a paddle, or a trackball. In each case, a direction is sent to the program which is used to redirect the movement of the animated image. There are essentially two approaches that can be used.

The device being used can be explicitly tested in the program for a change in its status, such as the cursor left key being pressed or the joystick being shifted downward. The testing must be done periodically during the programs execution. It's important that the tests are frequent enough so that the program is responsive to its players but not so often so as to cause unneeded overhead in processing.

The second approach involves what is known as "event trapping." This concept involves using the PC's advanced BASIC and essentially automates the testing for the keys being pressed by the game player. Alas, there is no event trapping for a shift in the joystick and other devices. The technique is most often found with the four cursor control keys and uses the ON KEY(n) GOSUB xxxx statement, where n may range from 1 to 14 for the ten function keys (1 to 10) and the four cursor control keys (11 to 14). The xxxx is the line number for the subroutine in the program. See Program Segment 8.1.

```
ON KEY (10) GOSUB 1500
```

This tells BASIC to do a GOSUB 1500 whenever function key 10 is pressed. Key numbers 12 and 13 refer to cursor left and right, and keys 11 and 14 refer to cursor up and down. BASIC under version 2 of DOS allows additional keys, numbered from 15 to 20. They may be assigned to any control, shift, or super-shift key.

Before the key is actually processed ("trapped"), it must be made active. For example, consider the following code in Program Segment 8.2.

```
ON KEY (1) GOSUB 9030

KEY (1) ON
```

The ON KEY assigns a subroutine at line 9030 to any depressing of function key number 1. The second statement, KEY ON, tells the

PC to be on the alert for this key. As the program progresses, the user may interrupt it by simply pressing F1. An automatic GOSUB 9030 is performed. The code beginning at line 9030 is processed. When a RETURN is executed, the program resumes whatever it was doing when it was interrupted. In particular, this is how the "abort" key code, which begins each of our games, is handled. This is a formidable capability in the PC's BASIC and will be put to good use in all the action games that follow.

Some additional statements are used in conjunction with the KEY ON statement. These include the KEY (n) STOP and KEY (n) OFF. They allow turning off the PC's checking when a key is pressed. The STOP option stops the checking altogether, and OFF does not allow an interrupt, but remembers if one occurs when the next KEY ON statement is processed. These statements are used whenever some critical code is being executed that must not be interrupted. Note that when the GOSUB is executed, an automatic KEY OFF is done for you for the key that was pressed. Hence, the subroutine cannot be interrupted by its own key being pressed.

Another important issue in animated games is timing and multiple image control. Timing for BASIC games is often a problem in making the game operate as fast as possible. Interpreted BASIC adds an enormous amount of overhead to the running of programs. Thus, one solution is to simply compile the programs. On the other hand, if programmed carefully with an eye to performance, the animation can be done with reasonably good results.

Then you might have trouble compiling a finely tuned, interpreted progam—now the game may be too fast to play. In either case, the placement of some delay loops at cerrtain key points can alleviate this problem. Delay loops can also be used to incorporate variable levels of playing difficulty. A longer delay at certain points makes for a slower and easier game, while shorter delays create a more difficult level of play.

In terms of controlling multiple images, the choreography of the game and the sequence in which images are changed must be considered. For instance, if an image is destroyed or exploded, then everything might pause while the steps involved in the explosion sequence take place. On the other hand, each image might have a set of status indicators that indicate what is next for that image. Each image is then processed in sequence, each image being handled in a single step-by-step fashion. This is illustrated in the asteroid attack

game, which can have a variable number of images moving about the screen at one time.

There are six games provided in this chapter. They run the gamut from the basic "shootem up" to a takeoff on that pioneer of computer games, "Pong," ending with a maze chase game.

Bust'em

The game of Bust 'em consists of keeping a bouncing ball in play while it breaks up a wall of blocks ("busts them apart") displayed at the top of the screen. You have a paddle that you may move left and right to bat the ball up into the barrier. Use the left and right cursor control keys to shift the paddle in either direction. If a ball misses the paddle, it flys by and is gone forever. The game is designed to let you decide how many balls you wish to play with. The game is won when the barrier has been completely destroyed. See Figure 8.1 for an illustration of the game being played.

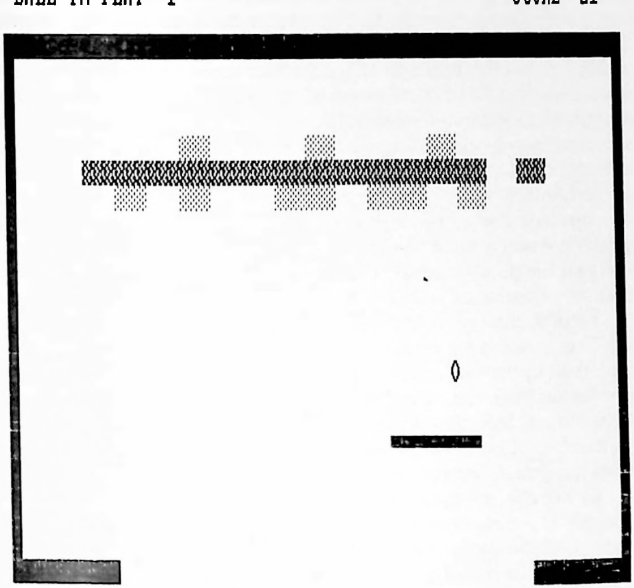
From a programming standpoint, Bust 'em is fairly straight-forward. The initial setup simply displays the overall playing area. The box with an opening at its bottom is first printed using the block character code 219 except for the opening, which uses the character code 0. This character appears on the screen as a blank and will be used when checking if a ball has bounced through at the bottom of the playing area.

The paddle uses the partial block character number 223. The barrier consists of three rows fo forty-five characters: The first- and third-row character codes are 176 and the second- or middle-row character codes are 178. The barrier is also printed using different colors to add some zest to the display. For users with a black and white display, the different character codes in the barrier make them stand out reasonably well.

Once the playing area is constructed, the game begins. The program "tosses" a ball into the center of the screen moving downward toward your paddle. In general, the bulk of the program consists of handling the ball for each of the various objects it might hit: your paddle, the barrier, the walls of the playing area, and the opening at the bottom of the playing area. Except for the opening, the ball must "bounce" or rebound off anything it hits.

The ball is moved by erasing it from its current position, calculating its new position, reprinting it at the new position. The process is

FIGURE 8.1. Bust'em game example.



then repeated until something other than a blank space is encountered in the new position.

The variables R and C maintain the ball's row and column on the screen. They are changed by the addition of row and column increments. To move diagonally down the screen from left to right, for instance, add 1 to both R and C. Likewise, to move up from the left to the right, add −1 to the row (it decreases) and +1 to the column. The variables RI and CI contain the increments for the four different directions the ball can take.

Thus, the program code is to simply add RI and CI to R and C for a new row and column. A blank is printed at the old R and C, the values are set to their new position, and the ball is reprinted. To check if the ball will hit anything after being moved, we use the SCREEN function. It returns the ASCII character value at a row and column position of the screen. For example, when the new row and

column place the ball into a wall, the SCREEN value returned will be a 219. If a 223 is returned, it means the paddle was hit, and so on.

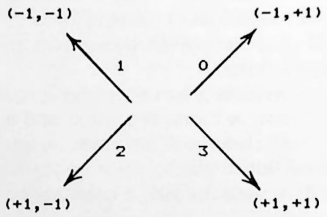
If a wall is hit, then the ball simply bounces off in the appropriate direction. This is also done for the ball hitting the paddle but with an added twist: If the ball hits either the first three or last three characters in the paddle (that is, its ends), then an extra lateral bounce is randomly added to the rebounding column coordinate. A value of 2 or 3 is randomly chosen and added to the ball's column value for a rightward bounce or subtacted for a leftward bounce.

When the ball hits the barrier, it erases three characters from it and updates the score. If the barrier has been completely erased (which is when a score of 45 has been achieved), then the player has won and the game is over; otherwise, the ball bounces off the barrier after the erasure, as with the walls and paddle.

Finally, the ball might get by the paddle and go into the opening. This is noted by encountering a zero character code. The program then updates a count of the number of balls used, checks if that was the last ball, and, if not, continues the game with the next one. If it was the last ball, then the game is over. The variable NB maintains the number of balls left in the game. It is initially set by the player when the game begins and is limited to from 1 to 9.

To handle the bounce of the ball, the variable D is used. It indicates the direction that the ball is moving at any point in the game. It also determines the row and column changes, or increments, used in moving the ball from one position to the next. The variable may take on values in the range of 0 to 3 for the directions shown in Figure 8.2. Based on the value of D, the row and column increments and the variables RI and CI are set for calculating the ball's new position.

FIGURE 8.2. Direction codes for moving ball.



When the ball hits something, it bounces. The variable D is also used to determine the direction from where the ball came. It is then reset to indicate the new direction that the ball takes in bouncing off the object it hit.

Consider the ball moving into the right side wall, and D having a value of 0. This indicates that the ball came in a diagonal lower-left to upper–rightward direction. The ball then rebounds in the direction number 3, or toward the upper left. Note that what is being hit must also be considered. If the top wall was hit with the same direction, then the bounce would be downward and to the right or toward direction number 1. One last check must be made. If the ball's new position after a change in direction is calculated is also a wall, then the ball has hit into a corner. The new direction is then changed to the opposite of the ball's previous direction.

The listing of Bust'em is Program 8.1.

```
0 '  BUSTM    "Bust" through a wall of block characters
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  Play an action game to break through the brick wall
50 '
60 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS : DEFINT A-Z
70 RANDOMIZE VAL(RIGHT$(TIME$,2)+MID$(TIME$,4,2)) : PLAY "MB L64"
80 PRINT "BUSTM  Bust'em through a wall" : PRINT
90 PRINT "How many balls in this game (1-9)? ";
100 NB=0 : LOCATE ,,1
110 WHILE NB<1 OR NB>9 : NB=VAL(INKEY$) : WEND : PRINT NB : PRINT
120 PRINT "Press <SPACE> to begin game."; : WHILE INKEY$<>" " : WEND
130 '
140 ' display playing area
150 '
160 CLS : LOCATE ,,0 ' clear screen and turn off cursor
170 LOCATE 1,12:PRINT "BALL IN PLAY" ; TAB(60) ; "SCORE" ;
180 LOCATE 3,11:PRINT STRING$(60,219); ' top of box
190 FOR R=4 TO 23
200 LOCATE R,11:PRINT CHR$(219);  ' left side
210 LOCATE R,70:PRINT CHR$(219);  ' right side
220 NEXT
230 LOCATE 24,11:PRINT STRING$(10,219);STRING$(40,0);STRING$(10,219); ' bottom
240 LOCATE 7,18 : COLOR 2 : PRINT STRING$(45,176); ' blocks
250 LOCATE 8,18 : COLOR 4 : PRINT STRING$(45,178); ' to break
260 LOCATE 9,18 : COLOR 6 : PRINT STRING$(45,176); ' through
270 '
280 ' initialize variables and subroutine traps
290 '
300 COLOR 7,0:PR=19:PC=37:P$=STRING$(9,223):LOCATE PR,PC:PRINT P$; ' paddle
310 ON KEY(12) GOSUB 570 : KEY(12) ON  ' cursor left - paddle shifts left
320 ON KEY(13) GOSUB 590 : KEY(13) ON  ' cursor right - paddle goes right
330 BH=0 : BP=0 : R=10 : C=40 : N=0 : GOTO 680 ' fake ball exit for start-up
340 '
```

```
350 '  main loop
360 '
370 LOCATE R,C : PRINT "O";  ' display ball
380 FOR I=1 TO 80 : NEXT   ' delay between moves
390 N=SCREEN (R+RI,C+CI)  ' what's at new position
400 IF N=32 THEN 520     ' blank, continue moving
410 IF N=0 THEN 680   ' null, got by paddle
420 IF N=178 OR N=176 THEN 800 ' hit blocks
430 PLAY "N50"    ' sound effect
440 VW=SCREEN(R+RI-1,C+CI)<>32 AND SCREEN(R+RI+1,C+CI)<>32 ' wall or paddle hit
450 IF VW THEN D=3-D ELSE IF D=1 OR D=3 THEN D=D-1 ELSE D=D+1 ' direction?
460 RI=-1 : IF D=1 OR D=2 THEN RI=1 ' new row and
470 CI=-1 : IF D=0 OR D=1 THEN CI=1 '  column increment
480 IF SCREEN(R+RI,C+CI)<>32 THEN VW=NOT VW : GOTO 450 ' don't move into wall
490 IF NOT (N=223 AND (C-PC<3 OR C-PC>5) ) THEN 520     ' ends of paddle hit?
500 LOCATE R,C : PRINT " ";      ' erase ball off paddle
510 IF D<2 THEN C=C+INT(2+RND*2) ELSE C=C-INT(2+RND*2) ' extra bounce shift
520 LOCATE R,C : PRINT " ";      ' erase old position
530 R=R+RI : C=C+CI : GOTO 370  ' set R and C for new and continue
540 '
550 '  subroutines to move the paddle, P$
560 '
570 IF PC<18 THEN RETURN ELSE KEY (13) OFF ' shift left
580 SH=-5 : KY=13 : GOTO 610
590 IF PC>56 THEN RETURN ELSE KEY (12) OFF ' shift right
600 SH=+5 : KY=12
610 RT=CSRLIN : CT=POS(0)          ' save ball's position
620 LOCATE PR,PC : PRINT SPC(9);   ' erase paddle
630 PC=PC+SH : LOCATE PR,PC : PRINT P$  ' re-display it
640 LOCATE RT,CT : KEY(KY) ON : RETURN  ' re-LOCATE for ball
650 '
660 ' ball is lost
670 '
680 BP=BP+1 : IF BP>NB THEN 750 ' all balls played?
690 LOCATE 1,25 : PRINT BP;      ' how many balls left
700 LOCATE R,C  : PRINT " ";     ' erase current ball
710 R=10 : C=40+INT(RND*3)       ' set row, column,
720 RI=1 : CI=0 : D=INT(2*RND+1) ' increment, and direction
730 FOR I=1 TO 500:C$=INKEY$:NEXT ' pause before new ball
740 GOTO 370
750 LOCATE 14,25 : PRINT "GAME OVER.  Sorry... you lose";
760 GOTO 870
770 '
780 ' blocks hit
790 '
800 BH=BH+1 : LOCATE 1,66 : PRINT BH; : PLAY "N35" ' up score
810 LOCATE R+RI,C+CI-((C+CI) MOD 3)  ' round column position
820 PRINT "   "; : IF BH<45 THEN 450 ' erase block (3 spaces)
830 '
840 '  end of game
850 '
860 LOCATE 14,25 : PRINT "GAME OVER.  Good going! You won!";
870 LOCATE R,C : PRINT " "; ' erase last ball
880 FOR I=1 TO 16 : C$=INKEY$ : NEXT ' clear any extra keystrokes
890 LOCATE 16,19: PRINT "Press Esc to exit or <SPACE> to play again.";
900 C$="" : WHILE C$="" : C$=INKEY$ : WEND
910 IF ASC(C$)=27 THEN 9050 ELSE IF C$=" " THEN 160 ELSE 900
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

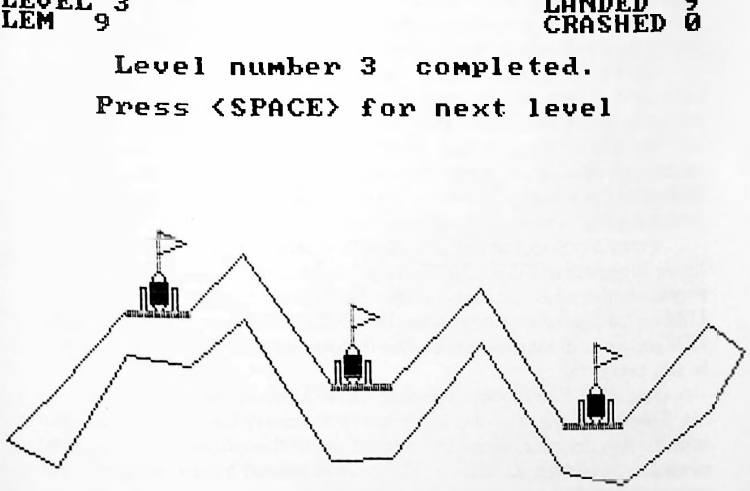
Program 8.1 BUSTM - Bust'em game

LEML

The game of LEML, Lunar Excursion Module Lander, allows you to pilot a LEM that is landing on the moon's surface. Cursor control keys move the module up, down, left, and to the right. This is a multi-screen game with three landing pads per screen. Each screen's pads are situated among some rugged terrain on the moon's surface. The pads get progressively smaller with each screen. As you fly the module, it must not touch any of the terrain other than the pads; otherwise, it crashes. There are four screens, each with three LEMs to land.

When you land a module, a flag is raised and a short tune is played. See Figure 8.3 for an illustration of the game being played where three LEMs have landed. A crash landing causes the LEM to explode. After three LEMs are piloted on a screen, the next screen is drawn and the process is repeated. The game ends after three LEMs have crashed or after the fourth screen is completed.

FIGURE 8.3. Lunar Lander game example.



This game illustrates many of the unique features in the PC's BASIC. The game is also written in the SCREEN 1 graphics mode and makes extensive use of the GET and PUT statements. They are very useful for animation in the grpahics modes because they allow complete images to be displayed and also erased with a single statement, as discussed in the opening of this chapter.

Of course, it is possible to write animation-based games using character-mode techniques. This is illustrated with a character mode version of LEML provided in the next section, called CHLEML (for CHaracter mode LEML).

In LEML, the movement of the LEM is done by constantly changing its position using point row and column increments. The increment values, stored in variables RI and CI, change based on the pressing of the cursor control keys. For instance, pressing the cursor-down key causes RI to be increased. The new position for the LEM when calculated will be at a lower point on the screen.

The four cursor control keys cause traps to four subroutines to change these increments. A check is also made to keep the values from going outside the range of −3 to +3. Thus, a LEM can, at most, change position by three points in any direction. Having a range of values also gives the effect of the LEM moving slower or faster as a cursor key is pressed repeatedly.

Before the game begins, the screen is cleared and the three animated objects used in the game are drawn in succession. After an object is displayed, a GET statement stores the image in an array: L for the LEM, F for the flag, and X for an explosion image. These arrays are then used in the various PUT statements that display the image as the game is played. Figure 8.4 illustrates the three images and the programming statements used in creating them.

From a programming standpoint, game play is controlled with three loops: One FOR / NEXT loop controls the changes from one screen to the next; a second FOR / NEXT loop keeps track of which LEM out of the three is active; and a WHILE / WEND loop controls the LEM moving about the screen. The three loops start at lines 370 to 420 in the program.

The WHILE loop simply moves the LEM about until something is hit. The coordinates of the LEM are maintained by the variables X and Y. The increments for movement, as in the game Bust 'em, are stored in variables XI and YI. They are changed by the cursor keys being pressed, as mentioned earlier. After a new position is calcu-

FIGURE 8.4. LEML image examples.

LEM

```
LINE  (4,4)-(11,12),3,BF : LINE (4,4)-(7,0)
LINE -(8,0) : LINE -(11,4) : LINE (4,12)-(7,15)
LINE -(8,15) : LINE -(11,12)
LINE  (0,6)-(2,15),3,B : LINE(13,16)-(15,15),3,B
```

EXPLOSION

```
PSET (7,5),3 : DRAW "C1 NU5 ND10 NL7 NR7 C2 NH4 NG4"
PSET (8,5) : DRAW "C1 NU5 ND10 NL7 NR7 C2 NE4 NF4"
```

FLAG

```
PSET (0,0),1 : DRAW "D15 R1 U15"
LINE (1,1)-(12,5),3 : LINE -(1,9),3
```

lated, it is determined if the LEM will be moved off the screen. If so, the new position is ignored; otherwise, the new position is updated in X and Y and the image is PUT there.

A check is then made to determine if something was hit: either a pad and the LEM has landed, or the terrain which causes the LEM to crash. This is done by using the POINT function at coordinate points, which represent the bottom of the LEM. The POINT function returns the color of the point being referenced. Thus, the areas that might be hit are color-coded as follows.

If a point is 0, for the background, then movement is okay to this new position. The value 3 indicates the terrain, and a 2 means the landing pads are being moved onto. A GOSUB is then made to the appropriate subroutine: either updating the score and displaying the flag, or exploding the LEM. The next LEM is then started unless three of them have crashed, which means the game is over.

As play begins at each level, the moon's terrain with the landing pads is created. This is done by randomly drawing a jagged line across the bottom third of the screen, allowing three flat spots for the pads. The pads are based on the level number and become smaller as the levels increase.

The two arrays R and C contain the twelve-point coordinates for the line that makes up the terrain. At array elements 2, 5, and 8, the next point-row values in the array are equal. This creates a horizontal line to represent the flat landing-pad areas. Their width is based on the level number. For all other array elements, a random position

is chosen as the next screen point. This gives the jagged effect of a mountainous terrain. Additional lines are drawn below each pad to give them some depth and emphasis.

The R and C arrays are used a second time to draw another line below the first one. The row points are decremented some random value to create another line below the first one. This adds further to the ruggedness of the terrain.

The game is listed in Program 8.2, LEML.

```
0 ' LEML  LEM Lander - as in Lunar Lander
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  Graphics mode version of the game lunar lander.
50 '
60 SCREEN 0,1 : COLOR 7,0 : WIDTH 40 : KEY OFF : CLS : DEFINT A-Z
70 RANDOMIZE VAL(RIGHT$(TIME$,2)+MID$(TIME$,4,2))
80 DIM R(11),C(11),L(33),X(33),F(33)
90 PRINT"LEML  Lunar Excursion Module Lander" : PRINT
100 DEF SEG=&H40 : IF (PEEK(&H10) AND &H30)<>&H30 THEN 120 ' correct adapter?
110 PRINT"Program requires color/graphics adapter." : GOTO 9030
120 PRINT "Press <SPACE> to begin game." : WHILE INKEY$<>" " : WEND
130 SCREEN 1,0 : COLOR 0,1 ' set graphics mode and palette
140 '
150 '   GET the LEM, L, crash explosion, X, and flag, F
160 '
170 CLS : LINE (4,4)-(11,12),3,BF : LINE (4,4)-(7,0) : LINE -(8,0):LINE -(11,4)
180 LINE (4,12)-(7,15) : LINE -(8,15):LINE -(11,12)
190 LINE (0,6)-(2,15),3,B : LINE (13,6)-(15,15),3,B
200 GET (0,0)-(15,15),L ' lunar module
210 CLS : PSET (7,5),3 : DRAW "C1 NU5 ND10 NL7 NR7 C2 NH4 NG4"
220 PSET (8,5) : DRAW "C1 NU5 ND10 NL7 NR7 C2 NE4 NF4"
230 GET (0,0)-(15,15),X ' explosion
240 CLS : PSET (0,0),1 : DRAW "D15 R1 U15"
250 LINE (1,1)-(12,5),3 : LINE -(1,9),3
260 GET (0,0)-(15,15),F ' landing flag
270 '
280 '  set cursor control trapping subroutines
290 '
300 ON KEY(11) GOSUB 690 : ON KEY(12) GOSUB 720 ' up, left,
310 ON KEY(13) GOSUB 750 : ON KEY(14) GOSUB 780 ' right, down
320 FOR K=11 TO 14 : KEY (K) ON : NEXT          ' turn keys on
330 TL=0 : LAN=0 : LCR=0 ' total LEMs, landed, and crashed
340 '
350 ' begin play
360 '
370 FOR LEV=1 TO 4    ' for each level
380 GOSUB 840         ' create a background
390   FOR LEM=1 TO 3 ' and fly 3 LEMs
400   TL=TL+1 : LOCATE 2,5 : PRINT TL;
410   P1=0 : P2=0 : R=0 : C=160 : RI=3 : CI=3-INT(RND*7)
420       WHILE P1=0 AND P2=0  ' while in space
430       PUT (C,R),L          ' display a LEM
440       RN=R+RI : CN=C+CI    ' change position
```

```
450       IF RN<0 THEN RN=0 ELSE IF RN>184 THEN RN=184
460       IF CN<0 THEN CN=0 ELSE IF CN>304 THEN CN=304
470       PUT (C,R),L  ' erase LEM
480       R=RN : C=CN  ' set new position
490       P1=POINT(C,R+15) OR POINT(C,R+14)       ' left pad hit?
500       P2=POINT(C+15,R+15) OR POINT(C+15,R+14) ' right pad hit?
510       WEND
520    IF P1=2 AND P2=2 AND RI<2 AND CI<2 THEN GOSUB 1020 ELSE GOSUB 1130
530    IF LCR=3 THEN 610 ' three crashes?
540    NEXT LEM
550 LOCATE 4,7 : PRINT "Level number";LEV;" completed."
560 FOR I=1 TO 10 : C$=INKEY$ : NEXT
570 IF LEV=4 THEN 610 ' skip message on last level
580 LOCATE 6,6 : PRINT "Press <SPACE> for next level"
590 WHILE INKEY$<>" " : WEND
600 NEXT LEV
610 LOCATE 6,6 : PRINT "GAME OVER.  Want to play more?  "
620 FOR I=1 TO 16 : C$=INKEY$ : NEXT ' clear any extra keystrokes
630 LOCATE 7,3 : PRINT "Press Esc to exit or SPACE for new game"
640 C$="" : WHILE C$="" : C$=INKEY$ : WEND
650 IF ASC(C$)=27 THEN 9050 ELSE IF C$=" " THEN 330 ELSE 640
660 '
670 ' cursor key traps; change Row, Column increments
680 '
690 RI=RI-1:IF RI<-3 THEN RI=-3:RETURN 'up
700 IF RI=0 AND CI=0 THEN RI=-1
710 RETURN
720 CI=CI-1:IF CI<-3 THEN CI=-3:RETURN 'left
730 IF CI=0 AND RI=0 THEN CI=-1
740 RETURN
750 CI=CI+1:IF CI>3 THEN CI=+3:RETURN  'right
760 IF CI=0 AND RI=0 THEN CI=+1
770 RETURN
780 RI=RI+1:IF RI>3 THEN RI=+3:RETURN  'down
790 IF RI=0 AND CI=0 THEN RI=+1
800 RETURN
810 '
820 '  display lunar terrain and landing pads
830 '
840 CLS : WID=(6-LEV)*8+2 : C(0)=0 : C(11)=319
850 PRINT "LEVEL" ; LEV ; TAB(30) ; "LANDED" ; LAN
860 PRINT "LEM" ; LEM ; TAB(30) ; "CRASHED" ; LCR
870 FOR I=1 TO 10 : C(I)=C(I-1)+RND*30+15 : NEXT ' column points
880 FOR I=0 TO 11 : R(I)=RND*75+100 : NEXT       ' and row
890 FOR I=3 TO 9 STEP 3 : C(I)=C(I-1)+WID : R(I)=R(I-1) : NEXT ' landings
900 PSET(C(0),R(0))
910 FOR I=1 TO 11:LINE -(C(I),R(I)):NEXT ' draw topmost terrain
920 FOR I=0 TO 2:FOR J=2 TO 8 STEP 3
930 LINE (C(J),R(J)+I)-(C(J+1),R(J+1)+I),2 ' draw landing pads
940 NEXT:NEXT
950 PSET (C(0),R(0)),2
960 FOR J=1 TO 10:LINE -(C(J),R(J)+RND*25+16):NEXT ' draw lower terrain
970 LINE -(C(11),R(11))
980 RETURN
990 '
1000 ' LEM lands okay
1010 '
1020 LAN=LAN+1 :  LOCATE 1,37 : PRINT LAN;
```

```
1030 PLAY "MB O4 d16 p16 c8 d16 p16 c8 d4" : PUT (C,R),L
1040 FOR R=R+6 TO R-15 STEP -1
1050 PUT (C+7,R),F ' raise flag
1060 FOR I=1 TO 50 : NEXT ' pause
1070 PUT (C+7,R),F ' erase flag
1080 NEXT
1090 PUT (C+7,R),F : RETURN ' final flag image
1100 '
1110 ' LEM crashes! explode it
1120 '
1130 LCR=LCR+1 : LOCATE 2,38 : PRINT LCR;
1140 PLAY "MB O2 D8 D8 C4"
1150 FOR I=1 TO 4
1160 PUT (C,R),X ' explosion
1170 FOR J=1 TO 99 : NEXT ' pause
1180 PUT (C,R),L ' re-display LEM
1190 NEXT
1200 FOR J=1 TO 199
1210 C$=INKEY$ ' pause and clear any extra keystrokes
1220 NEXT : RETURN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 LOCATE 25,1:PRINT SPC(39);
9040 LOCATE 25,1:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE RUN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 8.2 LEML - Lunar Excursion Module Lander

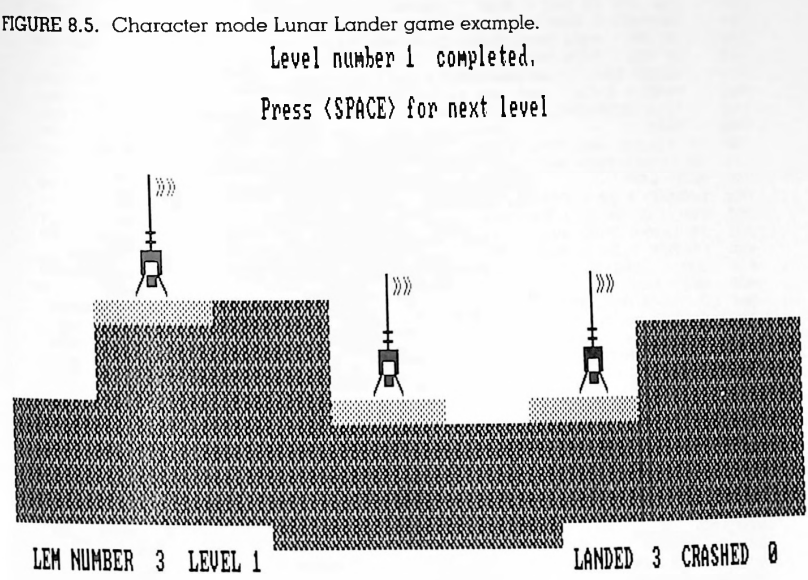
CHLEML

CHLEML is a character-made version of the graphics mode game LEML. The object of the game and overall play is the same, but a number of key programming steps vary quite a bit. The major difference is in how the images are moved and erased on the screen and the use of the PC's extended-character set to create the images. See Figure 8.5 for an illustration of the game. Program 8.3 is the listing of CHLEML.

Play of the game is identical to LEML. You have three LEMs to land per level. There are four levels in the game, each with smaller landing pads on which to land. The terrain of the landing surface also depicts a rugged landing surface that must be maneuverd over. The LEM is controlled by the four cursor control keys: Press these keys to move the LEM in the desired direction.

The image of the LEM, its erasure, and exploding it are all done via separate subroutines. Each subroutine contains COLOR,

FIGURE 8.5. Character mode Lunar Lander game example.




```
0 ' CHLEML  CHar mode LEM Lander - like LEML game
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  Character mode graphics version of the game lunar lander.
50 '
60 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS : DEFINT A-Z
70 RANDOMIZE VAL(RIGHT$(TIME$,2)+MID$(TIME$,4,2))
80 PRINT "CHLEML  Character mode Lunar Excursion Module Lander" : PRINT
90 PRINT "Press <SPACE> to begin game"; : WHILE INKEY$<>" ": WEND
100 '
110 ' set ON KEY subroutines
120 '
130 ON KEY(11) GOSUB 520 : ON KEY(12) GOSUB 550 ' up, left,
140 ON KEY(13) GOSUB 580 : ON KEY(14) GOSUB 610 ' right, down
150 FOR K=11 TO 14 : KEY (K) ON : NEXT           ' turn keys on
160 TL=0 : LAN=0 : LCR=0 : LOCATE ,,0
170 '
180 ' begin play
190 '
200 FOR LEV=1 TO 4   ' for each level
210 GOSUB 670        ' create background
220  FOR LEM=1 TO 3 ' and fly 3 LEMs
230  TL=TL+1 : LOCATE 25,14 : PRINT TL;
240  P1=32 : P2=32 : R=1 : C=39 : R1=1 : C1=1
```

```
250     WHILE P1=32 AND P2=32 ' while in space
260     GOSUB 1100           ' display a LEM
270     FOR I=1 TO 100 : NEXT ' pause
280     RN=R+RI : CN=C+CI     ' new position
290     IF RN<1 THEN RN=1 ELSE IF RN>19 THEN RN=19
300     IF CN<1 THEN CN=1 ELSE IF CN>74 THEN CN=74
310     GOSUB 1170  ' erase LEM
320     R=RN : C=CN ' set new position
330     P1=SCREEN(R+3,C) : P2=SCREEN(R+3,C+3)
340     WEND
350   IF P1=176 AND P2=176 THEN GOSUB 890 ELSE GOSUB 1000
360   IF LCR=3 THEN 440
370   NEXT LEM
380   LOCATE 5,27 : PRINT "Level number";LEV;" completed."
390   FOR I=1 TO 10 : C$=INKEY$ : NEXT
400   IF LEV=4 THEN 440 ' skip message on last level
410   LOCATE 7,26 : PRINT "Press <SPACE> for next level"
420   WHILE INKEY$<>" " : WEND
430   NEXT LEV
440 LOCATE 5,25 : PRINT "GAME OVER.  Want to play more?"
450 FOR I=1 TO 10 : C$=INKEY$ : NEXT
460 LOCATE 7,19 : PRINT "Press Esc to exit or <SPACE> for a new game."
470 C$="" : WHILE C$="" : C$=INKEY$ : WEND
480 IF ASC(C$)=27 THEN 9050 ELSE IF C$=" " THEN 160 ELSE 470
490 '
500 ' cursor key traps; change Row, Column increments
510 '
520 RI=RI-1:IF RI<-5 THEN RI=-5:RETURN 'up
530 IF RI=0 AND CI=0 THEN RI=-1
540 RETURN
550 CI=CI-1:IF CI<-8 THEN CI=-8:RETURN 'left
560 IF CI=0 AND RI=0 THEN CI=-1
570 RETURN
580 CI=CI+1:IF CI>8 THEN CI=8:RETURN   'right
590 IF CI=0 AND RI=0 THEN CI=+1
600 RETURN
610 RI=RI+1:IF RI>5 THEN RI=5:RETURN   'down
620 IF RI=0 AND CI=0 THEN RI=+1
630 RETURN
640 '
650 ' display terrain with landing pads
660 '
670 CLS : W=(6-LEV)*2+1 : WID=W  ' pad width
680 C(0)=1 : P=(80-WID*3)/8 ' initial column and half max valley width
690 FOR I=0 TO 6
700 R(I)=(RND/2)*SGN(RND-.5)*7+18
710 CHC=178 : WID=P*RND+P : COLOR 2,7
720 IF (I MOD 2)=1 THEN CHC=176 : WID=W : COLOR 6,7
730 C(I+1)=C(I)+WID
740 IF I=6 THEN C(7)=79 : WID=79-C(6)
750 LOCATE R(I),C(I) : PRINT STRING$(WID,CHC); ' landing pad or top
760 IF R(I)>23 THEN 800
770 FOR R=R(I)+1 TO 24
780   LOCATE R,C(I) : PRINT STRING$(WID,178); ' terrain below pad
790   NEXT R
800 NEXT I
810 COLOR 12,0
820 LOCATE 25,3 : PRINT "LEM NUMBER     LEVEL"; LEV;
830 LOCATE 25,55: PRINT "LANDED ";LAN;" CRASHED ";LCR;
840 LOCATE 24,1 : PRINT SPC(25); : LOCATE 24,54 : PRINT SPC(25);
850 COLOR 7,0 : RETURN
```

```
860 '
870 ' LEM lands okay
880 '
890 LAN=LAN+1 : LOCATE 25,62 : PRINT LAN;
900 GOSUB 1100 : PLAY "MB O4 d16 p16 c8 d16 p16 c8 d4"
910 COLOR 13,0
920 LOCATE R-2,C+1 : PRINT CHR$(179)+STRING$(2,175);
930 LOCATE R-1,C+1 : PRINT CHR$(179);
940 COLOR 7,0
950 FOR J=1 TO 199 : C$=INKEY$ : NEXT ' pause
960 RETURN
970 '
980 ' LEM crashes!
990 '
1000 LCR=LCR+1 : LOCATE 25,75 : PRINT LCR;
1010 PLAY "MB O2 D8 D8 C4"
1020 FOR I=0 TO 3
1030 ON (I MOD 2)+1 GOSUB 1240,1320
1040   FOR J=1 TO 40 : C$=INKEY$ : NEXT ' pause
1050 NEXT I
1060 GOSUB 1170 : RETURN
1070 '
1080 ' lunar module
1090 '
1100 LOCATE R,C  :PRINT " "+CHR$(216)+" ";
1110 LOCATE R+1,C:PRINT CHR$(222)+CHR$(223)+CHR$(221);
1120 LOCATE R+2,C:PRINT "/"+CHR$(223)+"\";
1130 RETURN
1140 '
1150 ' erasure for module and exposions
1160 '
1170 LOCATE R+0,C:PRINT "   ";
1180 LOCATE R+1,C:PRINT "   ";
1190 LOCATE R+2,C:PRINT "   ";
1200 RETURN
1210 '
1220 ' explosion part 1.
1230 '
1240 COLOR 4,7
1250 LOCATE R,C  :PRINT "X X";
1260 LOCATE R+1,C:PRINT " * ";
1270 LOCATE R+2,C:PRINT "X X";
1280 COLOR 7,0 : RETURN
1290 '
1300 ' explosion part 2.
1310 '
1320 COLOR 7,4
1330 LOCATE R,C  :PRINT "+ +";
1340 LOCATE R+1,C:PRINT " + ";
1350 LOCATE R+2,C:PRINT "+ +";
1360 COLOR 7,0 : RETURN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 8.3 CHLEML - Character mode LEML game

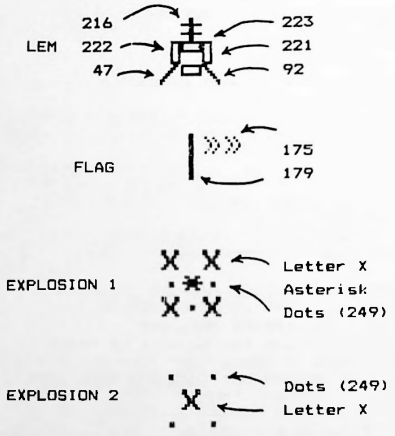
LOCATE, and PRINT statements for the appropriate image. Thus, the PUT statements used in LEML are replaced by GOSUBs. The LEM is displayed by a GOSUB to line 1100 and erased by the subroutine at line 1170. This erasure is also used when the LEM is exploded.

For additional effects, two explosion patterns are used when the LEM crashes. These are displayed by the subroutines at lines 1240 and 1320. Finally, the flag display is done in the subroutine when the LEM lands (there is no separate flag display subroutine). Figure 8.6 illustrates the four images used in this game and the character codes for each.

The movement of the character mode LEM is controlled in the same manner as in the graphics mode version of the program. The cursor control keys change row and column increments used in calculating the LEM's new position. The values in this case refer to character row and column distances, with a row increment being limited to the range of -5 to $+5$ and the column increment limited to -8 to $+8$ character positions.

Creating the background terrain is done using the shading character code number 176 to indicate a landing pad and the character code 178 for the rest of the terrain. Seven columns of characters are printed, with the second, fourth, and sixth columns

FIGURE 8.6. CHLEML image examples.



representing the width of the landing pad for the terrain in this level of play. The other four columns are some random width in size. This is similar to the manner in which the terrain was handled in the graphics mode LEML program.

Areas for the scoring are also displayed below the terrain, as opposed to above the game in LEML. This simplifies the problem of the LEM moving into the score if it was displayed at the top of the screen.

Controlling the overall flow of the game is identical to the techniques used in LEML. The counterparts of the two FOR / NEXT loops and the WHILE / WEND loop described in the LEML program preceding are in the character mode version. The only difference is that the checks for the LEM moving into a pad or the terrain consist of using the SCREEN function instead of POINT. A LEM may move freely into any area that is blank, a character code of 32. Any other character codes found, such as 176 and 178, cause the innermost loop to stop indicating that the LEM has landed or crashed. Processing in each case is similar to the LEML program.

When the LEM lands, a short tune is played, a flag is raised, and the score is updated. If the LEM crashes, the explosion subroutines are called and the LEM is then erased. Then the score is changed and the next LEM, if any, is started.

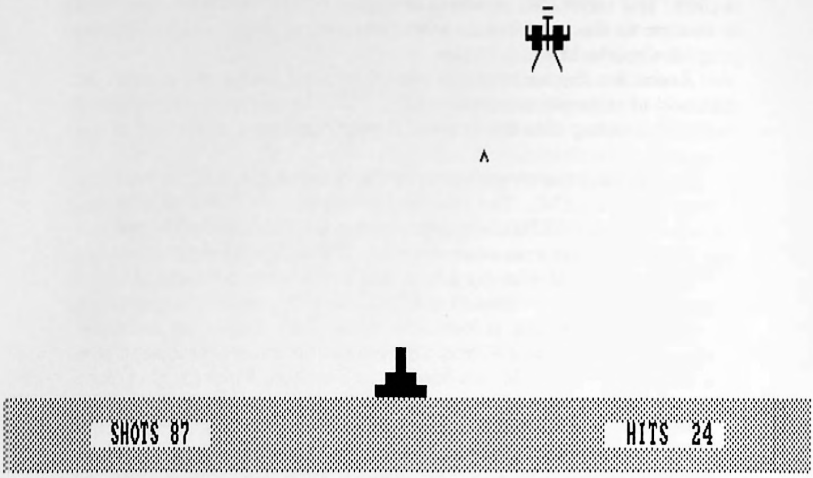
Satellite Invasion

The game of SATIN or Satellite Invasion is your basic "shoot 'em up" graphics game. A satellite flys across the screen and you must shoot it down. You enter the number of shots you would like to have, from 10 to 99. As you shoot at a satellite, each hit awards you with a new shot, so scores can be as high as you are accurate.

Your shooting apparatus is a cannon at the bottom of the screen, which you can move left or right to any one of five positions with the cursor left and cursor right keys. To fire the cannon, press the cursor-up key. A projectile is then shot out and moves vertically up the screen. Only one projectile may be active at a time. See Figure 8.7 for an example of the game being played.

The main loop in the program moves the satellite across the screen. A random row is initially calculated along with a random row increment to use in changing the satellite's position. A column

FIGURE 8.7. Satellite Invasion game.

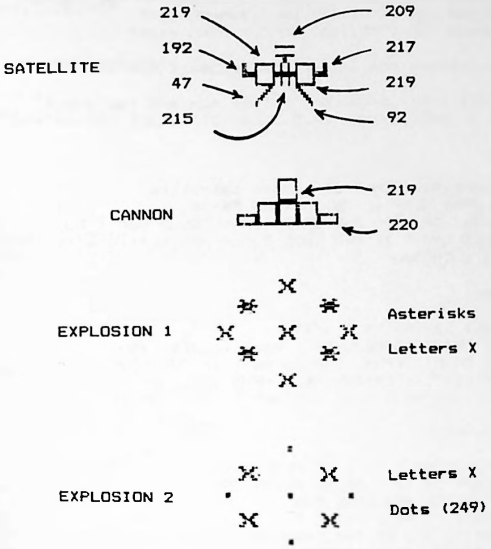


increment of −1 or +1 is used to move the satellite from the left or right side of the screen. A row increment value is set between −1 and +1 to give the satellite a diagonal up or down movement.

The game uses much of the programming developed in the games Bust 'em and the character mode version of LEML. The major difference is that there are three objects moving about: The cannon may change position, a satellite moves across the screen, and a projectile might be moving up the screen (after the cannon is fired). Figure 8.8 illustates the four images that may be displayed by the game. Note that two explosion images are used as in the CHLEML game for better effects when the satellite is hit. The projectile is the single character^, which is sometimes called a carat. Program 8.4 is the complete listing of the game.

Moving the cannon is similar to moving the paddle in Bust 'em. The cannon has five possible different positions. When moved, the previous position is erased when blanks are printed over it, variables for the new position are set, and the cannon is displayed again. The functions POS(0) and CSRLIN are used to maintain a copy of the cursor's position prior to moving the cannon. This is done to keep

FIGURE 8.8. Satellite Invasion game image examples.




```
0 ' SATIN  Satellite Invasion shoot'em up
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  Shoot down the satellites as they fly overhead.
50 '
60 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
70 DEFINT A-Z : DEFSNG R
80 RANDOMIZE VAL(RIGHT$(TIME$,2)+MID$(TIME$,4,2))
90 PRINT "SATIN - Satellite Invasion" : PRINT
100 LINE INPUT "Enter number of shots allowed (10-99)? ",N$
110 SH=VAL(N$) : IF SH<10 OR SH>99 THEN PRINT "From 10 to 99 only!" : GOTO 100
120 PRINT "Press <SPACE> to begin game." : WHILE INKEY$<>" " : WEND
130 TS=SH : TH=0 : P=0  ' total shots, total hits, projectile status
140 '
150 '  initialize screen, set ON KEY subroutines
160 '
170 COLOR 12,0 : CLS
180 FOR R=23 TO 25
190 LOCATE R,1 : PRINT STRING$(79,176); ' base
200 NEXT
210 COLOR 7,0
220 LOCATE 24,10:PRINT USING "  SHOTS ##  ";TS;
230 LOCATE 24,60:PRINT"  HITS     ";
```

```
240 CR=21 : CC=38 : GOSUB 910 ' initial cannon display
250 ON KEY(11) GOSUB 630 : KEY(11) ON ' fire button for cannon
260 ON KEY(12) GOSUB 670 : KEY(12) ON ' cannon left
270 ON KEY(13) GOSUB 720 : KEY(13) ON ' cannon right
280 '
290 ' starting satellite row,column and movement increments
300 '
310 R=INT(RND*16)+1 : RI=.1-RND/5  ' start row and increment
320 C=1 : CI=1 : IF RND<.5 THEN C=72 : CI=-1 ' start column and increment
330 '
340 ' main loop
350 '
360 GOSUB 750 : RN=R+RI : CN=C+CI ' move satellite
370 IF RN<1 THEN RN=1 ELSE IF RN>16 THEN RN=16
380 IF CI<0 AND CN<1 OR CI>0 AND CN>73 THEN GOSUB 860 : GOTO 310 ' off screen?
390 IF P THEN GOSUB 520 : IF R=0 THEN 310 ' projectile active?
400 R=RN : C=CN : GOTO 360
410 '
420 ' end of game
430 '
440 FOR I=1 TO 10 : C$=INKEY$ : NEXT
450 LOCATE 1,25 : PRINT "GAME OVER.  Want to play again?  "
460 LOCATE 3,19 : PRINT "Press Esc to exit or <SPACE> for new game..."
470 C$="" : WHILE C$="" : C$=INKEY$ : WEND
480 IF ASC(C$)=27 THEN 9050 ELSE IF C$=" " THEN 130 ELSE 470
490 '
500 ' process projectile
510 '
520 LOCATE PR,PC : PRINT " "; : IF PR>2 THEN 550 ' off screen?
530 TS=TS-1 : LOCATE 24,18 : PRINT USING "##"; TS; ' missed satellite
540 IF TS=0 THEN RETURN 440 ELSE 590
550 PR=PR-2 : LOCATE PR,PC : PRINT "^";
560 IF NOT (PR>INT(R) AND PR<INT(R+4) AND PC>C AND PC<INT(C+4)) THEN RETURN
570 GOSUB 1090 : GOSUB 1180 : GOSUB 860 : R=0   ' hit, explode satellite
580 TH=TH+1 : LOCATE 24,68 : PRINT USING "##"; TH; ' and increment score
590 KEY(11) ON : P=0 : RETURN  ' re-set projectile flag
600 '
610 ' fire button trap
620 '
630 KEY(11) OFF : P=1 : PR=20 : PC=CC+2 : RETURN ' start projectile
640 '
650 ' cannon moves left
660 '
670 KEY (13) OFF : IF CC>6 THEN CM=-16 : GOSUB 1010
680 KEY (13) ON : RETURN
690 '
700 ' cannon moves right
710 '
720 KEY (12) OFF : IF CC<70 THEN CM=+16 : GOSUB 1010
730 KEY (12) ON : RETURN
740 '
750 ' satellite
760 '
770 LOCATE R+0,C:PRINT"        ";
780 LOCATE R+1,C:PRINT"   "+CHR$(209)+"   "
790 LOCATE R+2,C:PRINT" "+CHR$(192)+CHR$(219)+CHR$(215)+CHR$(219)+CHR$(217)+" ";
```

```
800 LOCATE R+3,C:PRINT"   / \   ";
810 LOCATE R+4,C:PRINT"         ";
820 RETURN
830 '
840 ' satellite and explosion erasure
850 '
860 FOR I=0 TO 4
870 LOCATE R+I,C:PRINT"         ";
880 NEXT
890 RETURN
900 '
910 ' cannon
920 '
930 COLOR 14
940 LOCATE CR+0,CC:PRINT "  "+CHR$(219)+"  ";
950 LOCATE CR+1,CC:PRINT CHR$(220)+STRING$(3,219)+CHR$(220);
960 COLOR 7
970 RETURN
980 '
990 ' move cannon CM columns
1000 '
1010 TR=CSRLIN : TC=POS(0)
1020 LOCATE CR+0,CC:PRINT"       ";
1030 LOCATE CR+1,CC:PRINT"       ";
1040 CC=CC+CM : GOSUB 930
1050 LOCATE TR,TC : RETURN
1060 '
1070 ' explosion part 1.
1080 '
1090 LOCATE R+0,C:PRINT"    X    ";
1100 LOCATE R+1,C:PRINT" *    * ";
1110 LOCATE R+2,C:PRINT"X  X  X";
1120 LOCATE R+3,C:PRINT" *    * ";
1130 LOCATE R+4,C:PRINT"    X    ";
1140 RETURN
1150 '
1160 ' explosion part 2.
1170 '
1180 LOCATE R+0,C:PRINT"    *    ";
1190 LOCATE R+1,C:PRINT" x    x ";
1200 LOCATE R+2,C:PRINT"*  *  *";
1210 LOCATE R+3,C:PRINT" x    x ";
1220 LOCATE R+4,C:PRINT"    *    ";
1230 RETURN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 8.4 SATIN - Satellite invasion game

from disturbing the movement of the satellite and projectile on the screen before the cannon is repositioned. The cursor is then reset to its original position with a LOCATE after the cannon display is completed.

The main loop for moving the satellite also checks if movement of the projectile is required. The variable P is used to indicate whether or not the cannon has been fired. A zero means the projectile is not active; if P is not a zero, it indicates that the projectile should be moved along with the satellite. When the projectile is moved, its row value is decremented as it is displayed, causing it to move up the screen. Each time the projectile is printed, the program also checks if the satellite is hit and if the projectile is about to go off the top edge of the screen.

The projectile always starts at row 21 but at a column that is determined by the position of the cannon when fired by pressing the cursor-up key. The trapping subroutine simply sets P to -1, and the projectiles row and column variables PR and PC to their appropriate values. Note that the trapping routine also turns key number 11 off so that only one projectile can be active at a time. If this were not done, then subsequent depressions of the cursor-up key would reset P and a new projectile would be started, leaving the other one on the screen.

In order to activate more than one projectile at a time, you would have to keep track of which ones have been fired and their current positions on the screen. Furthermore, as each projectile is moved, it must be checked to determine if it hit the satellite or went off the screen. Programming for the animation of a variable number of objects requires the use of arrays that maintain information such as this for each object being manipulated. Our next game program illustrates this concept further.

Asteroid Attack

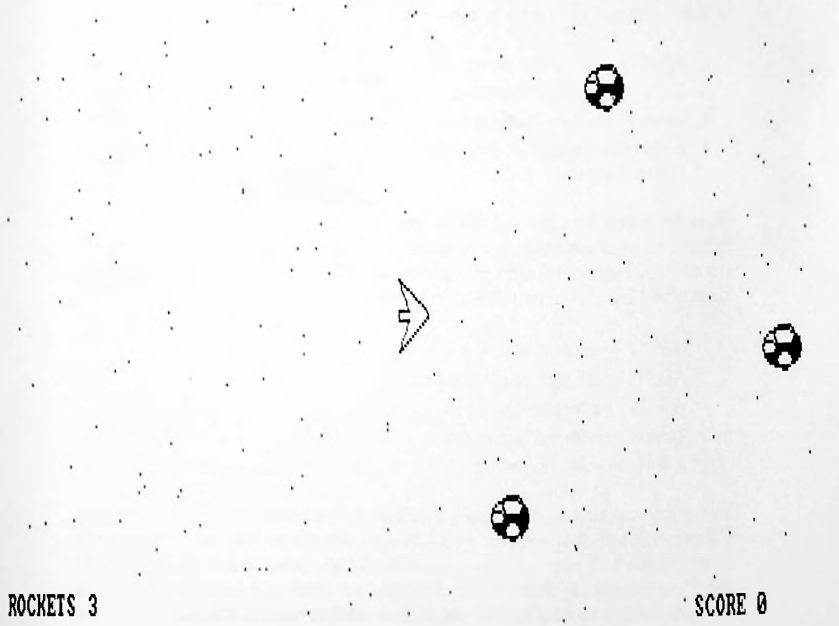
The game of Asteroid Attack is a high resolution, SCREEN 2, graphics mode "shoot 'em up" game. The overall thrust of the game is to pilot a space craft through a shower of asteroids which you destroy by shooting. After one asteroid explodes, another appears until there are up to six asteroids on the screen. You are allowed to move your rocket up, down, left, and right, using the four cursor control keys.

You shoot your "laser beam" by pressing Function key 10 (your trigger). If you collide with an asteroid, your space craft is destroyed. You are allowed three rockets; after these have been used, the game is over. Figure 8.9 illustrates the game being played.

Note that the limit of three rockets and six asteroids is controlled by the variables NR and MAXA. Set them to higher values, if desired. The initial number of asteroids that appear when the game is started is determined by the INA variable: It is initially set at 2.

This game essentially builds on the previous animated games but with a different approach to controlling the overall movement of the images. First, there may be up to seven objects on the screen (the rocket plus six asteroids), which is more than double the number handled in the previous games. Furthermore, each object must be

FIGURE 8.9. Asteroid Attack game.



controlled so that processing it does not slow down or hinder the actions of the other objects. This processing must also be in the proper sequence so that the action in the game occurs at a smooth pace.

Essentially, the game processes each object based on the object's current status—which can change, depending on player actions and the overall flow of the game. For example, an asteroid begins in a state of basic movement across the screen. A subroutine then processes the asteroid by moving it, depending on its current direction. On the other hand, if the player shoots at the asteroid and hits it, then the asteroid's state changes to "exploding." A different subroutine then processes the asteroid for the explosion. A fourth state is after the asteroid has exploded, at which point it is erased from the screen and reset to its initial status. The array AS maintains the status of each of the asteroids. At AS(i,0), we have the current status of asteroid i, which may take on the following values:

1. Initial startup of the asteroid at the edge of the screen.
2. Continue moving the asteroid in its current direction.
3. Laser beam hit; start exploding the asteroid.
4. Finish the explosion; erase the asteroid from the screen and reset the status back to 1.

Also contained in the AS array are the point positions and row and column increments for each asteroid. These values are used when an asteroid is moved across the screen. Thus, asteroid i is completely described by the array AS as follows:

- AS(i,0) = current status
- AS(i,1) = column point position
- AS(i,2) = row point position
- AS(i,3) = column increment
- AS(i,4) = row increment

The player's rocket also goes through a number of different states. Initially, the rocket moves in whatever direction the last cursor key was pressed. A second state consists of the rocket's laser beam having been fired. A third state is when an asteroid collides with the rocket, which results in the explosion of the rocket. Concluding the

explosion of the rocket is a fourth state (final erasure) that the rocket may take. The variable RS maintains these status values:

1. Initial startup of the rocket.
2. Move the rocket in its current direction.
3. Rocket fires laser beam at asteroid.
4. Asteroid collides with rocket; start explosion sequence.
5. Finish exploding rocket; erase from screen and reset status back to 1, unless this is the last rocket.

By using this status-based approach, a single WHILE / WEND loop controls the overall game flow. It simply processes each of the objects by examining the status of each and performing a GOSUB to the appropriate subroutine. This is continually done as long as there is a rocket left to fly in the game. The loop appears in the game as shown in Program 8.3, where the variable NR is the count of rockets and INA is the count of the number of active asteroids. See Program Segment 8.3.

```
510 WHILE NR>0
520   FOR I=1 TO INA
530   ON    RS   GOSUB 700,720,810,860,910 ' process rocket
540   ON AS(I,0) GOSUB 1010,1080,1130,1160  ' and astroids
550   NEXT
560 WEND
```

First the rocket is processed, and then one of the active asteroids is processed. While in a subroutine, an object's status may change. When this happens, a different subroutine will be used by the ON / GOSUB statements the next time the object is processed.

This also allows all objects to be processed in a step-by-step fashion. For example, instead of stopping everything to explode an asteroid, the space craft can continue moving while the explosion of the asteroid continues in sequence with everything else. Indeed, there may be two or three asteroids exploding at once!

The animation of each object is done with the now-familiar PUT statement. After each image is initially drawn, a GET statement saves it in an array: R contains the rocket, A contains the image of an asteroid, X contains an explosion pattern to use when either the rocket or asteroid is hit, and S contains an image of the laser being fired. See Figure 8.10 for an illustration of each image.

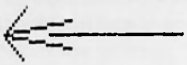
As a backdrop to our game, the screen has 200 random points set with the PSET statement. This gives the effect of stars. Since all

FIGURE 8.10. Astroid game rocket, asteroid, explosion, and laser beam.

ROCKET

```
DRAW "BM0,0 M23,10 D2 M0,23 M8,13 L8 U3 R8 M0,0"
```

LASER BEAM



```
DRAW "BM0,8 NM+16,-8 NM+48,-4 NM+48,+4
      NM+16,+8 BR32 R95"
```

ASTEROID

```
CIRCLE (6,16),5 : CIRCLE (18,12),8 : CIRCLE (18,20),6
      CIRCLE (16,16),15 : PAINT (16,16)
```

EXPLOSION 1

```
DRAW "BM10,14 M15,0 M20,14 M31,15 M20,16
      M15,31 M10,16 M0,15 M10,14"
DRAW "BM15,15 NU8 ND8 NL8 NR8 NE8 NF8 NG8 NH8"
```

EXPLOSION 2

```
DRAW "BM10,14 M15,0 M20,14 M31,15 M20,16 M15,31
      M10,16 M0,15 M10,14" : PAINT (15,15)
```

image movement is controlled with the PUT statement using the XOR parameter (by default), this background is not disturbed. Thus, planets and more complex-looking stars could just as easily be used as a backdrop without being upset by the other images being moved over them. Consider doing this in a character mode game. It would be very difficult to maintain and control a complicated background using the usual COLOR, LOCATE, and PRINT statement sequence. Program 8.5 contains the complete listing for the game.

```
0 '  ASTROID  Asteroid attack "shoot'em up" game
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 ' Rocket shoots asteroids with F10 key or Tab key.
50 ' Use cursor keys to move rocket about screen.
60 '
70 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS : DEFINT A-Z
80 RANDOMIZE VAL(RIGHT$(TIME$,2))
90 DIM R(37),RS(5),A(65),AS(7,4),S(129),E1(65),E2(65)
100 PRINT"ASTROID  -  Asteroid Attack":PRINT
110 DEF SEG=&H40 : IF (PEEK(&H10) AND &H30)<>&H30 THEN 130 ' correct adapter?
120 PRINT"Program requires color/graphics adapter." : GOTO 9030
130 PRINT"Press <SPACE> to begin game" : WHILE INKEY$<>" ": WEND
140 SCREEN 2 ' set mode to high resolution
150 '
160 ' GET rocket, asteroids, and explosions
170 '
180 CLS:DRAW "BM0,0M23,10 D2 M0,23 M8,13 L8 U3 R8 M0,0"
190 GET (0,0)-(23,23),R ' rocket
200 CLS:CIRCLE (6,16),5 : CIRCLE (18,12),8 : CIRCLE (18,20),6
210 CIRCLE (16,16),15 : PAINT (16,16)
220 GET (0,0)-(31,31),A ' asteroid
230 CLS:DRAW"BM0,8 NM+16,-8 NM+48,-4 NM+48,+4 NM+16,+8 BR32 R95"
240 GET (0,0)-(127,15),S ' shooter
250 CLS:DRAW "BM10,14 M15,0 M20,14 M31,15 M20,16 M15,31 M10,16 M0,15 M10,14"
260 DRAW "BM15,15 NU8 ND8 NL8 NR8 NE8 NF8 NG8 NH8"
270 GET (0,0)-(31,31),E1 ' 1st explosion
280 CLS:DRAW "BM10,14 M15,0 M20,14 M31,15 M20,16 M15,31 M10,16 M0,15 M10,14"
290 PAINT (15,15)
300 GET (0,0)-(31,31),E2 ' 2nd explosion
310 CLS : PLAY "MB MS L64"
320 FOR I=1 TO 200
330 PSET (RND*640,RND*200) ' background stars
340 NEXT
350 '
360 '  init variables and key traps
370 '
380 INA=2 : MAXA=6  ' initial & max number asteroids in game
390 NR=3  : RS=1     ' number of rockets, initial status
400 FOR I=1 TO INA : AS(I,0)=1 : NEXT ' asteroid statuses
410 ON KEY (10) GOSUB 1210 : KEY (10) ON ' laser
420 ON KEY (11) GOSUB 1460 : KEY (11) ON ' up
430 ON KEY (12) GOSUB 1470 : KEY (12) ON ' left
440 ON KEY (13) GOSUB 1480 : KEY (13) ON ' right
450 ON KEY (14) GOSUB 1490 : KEY (14) ON ' down
460 LOCATE 25,1:PRINT"ROCKETS";NR;:LOCATE 25,68:PRINT"SCORE 0";
470 '
480 ' Main loop
490 '
500 LOCATE 1,1
510 WHILE NR>0
520   FOR I=1 TO INA
530   ON    RS   GOSUB 700,720,810,860,910 ' process rocket
540   ON AS(I,0) GOSUB 1010,1080,1130,1160  ' and astroids
550   NEXT
```

```
560 WEND
570 '
580 ' end of game
590 '
600 LOCATE 1,1 : PRINT SPC(79) : PRINT SPC(79)
610 LOCATE 1,28: PRINT "GAME OVER. Want to play again?"
620 FOR I=1 TO 10 : C$=INKEY$ : NEXT
630 LOCATE 2,19: PRINT "Press Esc to exit or <SPACE> for a new game..."
640 C$="" : WHILE C$="" : C$=INKEY$ : WEND
650 IF ASC(C$)=27 THEN 9050 ELSE IF C$=" " THEN 310 ELSE 640
660 '
670 '  rocket subroutines
680 '
690 ' initial start-up of rocket
700 RX=300:RY=90:RS=2:X=0:Y=0:PUT(RX,RY),R:RETURN
710 ' move rocket
720 NX=RX+X : NY=RY+Y
730 IF NX<0 THEN NX=0 ELSE IF NX>487 THEN NX=487 ' off screen?
740 IF NY<0 THEN NY=0 ELSE IF NY>175 THEN NY=175
750 PUT(RX,RY),R : PUT(NX,NY),R : RX=NX : RY=NY : SF=0
760 FOR J=1 TO INA
770 IF AS(J,0)=2 THEN GOSUB 1360 : IF SF THEN RETURN ' collided with asteroid?
780 NEXT
790 RETURN
800  ' start exploding rocket
810 FOR J=1 TO 5
820 PUT(RX,RY),R : PUT(RX,RY),E2 ' explosion
830 NEXT
840 RS=4 : RETURN
850 ' finish exploding rocket
860 PUT(RX,RY),E1:PUT(RX,RY),E2:PUT(RX,RY),E1
870 NR=NR-1 : LOCATE 25,8: PRINT NR; ' one less rocket
880 FOR K=10 TO 14 : KEY (K) ON : NEXT
890 RS=1 : RETURN
900 ' rocket fires laser beam
910 IF RS=3 OR RS=4 THEN RETURN ' exploding?
920 PUT (RX+24,RY+3),S : SF=SC
930 FOR J=1 TO INA
940 IF AS(J,0)=2 THEN GOSUB 1270 : IF SF<>SC THEN 960 ' hit asteroid?
950 NEXT
960 PUT (RX+24,RY+3),S : RS=2 : RETURN
970 '
980 '  asteroid subroutines
990 '
1000 ' start an asteroid
1010 ID=I MOD 4 : AS(I,0)=2 : XA=RND*480+80 : YA=RND*100+40
1020 IF ID=1 THEN AS(I,1)=XA :AS(I,2)=0 :AS(I,3)=8  :AS(I,4)=12 :GOTO 1060
1030 IF ID=2 THEN AS(I,1)=590:AS(I,2)=YA:AS(I,3)=-12:AS(I,4)=0  :GOTO 1060
1040 IF ID=3 THEN AS(I,1)=XA :AS(I,2)=150:AS(I,3)=0 :AS(I,4)=-12:GOTO 1060
1050 IF ID=0 THEN AS(I,1)=0  :AS(I,2)=YA :AS(I,3)=8 :AS(I,4)=-12
1060 PUT(AS(I,1),AS(I,2)),A : RETURN
1070 ' move asteroid
1080 NXA=AS(I,1)+AS(I,3) : NYA=AS(I,2)+AS(I,4) : ASD=2
1090 IF NXA<0 OR NXA>591 OR NYA<0 OR NYA>151 THEN ASD=1 ' off screen?
1100 PUT(AS(I,1),AS(I,2)),A : IF ASD=2 THEN PUT(NXA,NYA),A
1110 AS(I,1)=NXA:AS(I,2)=NYA:AS(I,0)=ASD:RETURN
1120 ' explode asteroid
1130 PUT(AS(I,1),AS(I,2)),A : PUT(AS(I,1),AS(I,2)),E1
```

```
1140 AS(I,0)=4 : RETURN
1150 ' end explosion
1160 AS(I,0)=1 : PUT(AS(I,1),AS(I,2)),E1
1170 PUT(AS(I,1),AS(I,2)),E2:PUT(AS(I,1),AS(I,2)),E2
1180 IF INA<MAXA THEN INA=INA+1 : AS(INA,0)=1 'next asteroid status
1190 RETURN
1200 '
1210 ' trap laser
1220 '
1230 RS=5 : PLAY "N71 N72 N74 N78" : RETURN
1240 '
1250 ' laser hit asteroid?
1260 '
1270 AX=AS(J,1) : AY=AS(J,2)
1280 IF AX<RX+24 OR AX>RX+151 THEN RETURN
1290 IF AY<RY-12 OR AY>RY+4   THEN RETURN
1300 SC=SC+1 : LOCATE 25,73 : PRINT SC; ' up score
1310 PLAY "L32 N32 N48 N48 L64" : AS(J,0)=3  ' explode status
1320 RETURN
1330 '
1340 ' asteroid hit rocket?
1350 '
1360 IF RS=3 THEN RETURN ' already hit?
1370 AX=AS(J,1) : AY=AS(J,2)
1380 IF AX<RX-15 OR AX>RX+8 THEN RETURN
1390 IF AY<RY-15 OR AY>RY+8 THEN RETURN
1400 FOR K=10 TO 14 : KEY (K) OFF : NEXT
1410 PLAY "L16 N24 N16 L8 N8 L64"
1420 RS=3 : SF=1 : RETURN ' explode status
1430 '
1440 ' traps for rocket direction changes
1450 '
1460 X=0  :Y=-8:RETURN
1470 X=-16:Y=0 :RETURN
1480 X=+16:Y=0 :RETURN
1490 X=0  :Y=+8:RETURN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 8.5 ASTROID - Asteroid attack game

Maze Chase

Our last action game is Maze Chase. This is the familiar "eat 'em up" game where you move a figure through a maze, eating dots and evading monsters that chase you. You are represented in the game

by the smiley-faced character. For monsters, we use three foreign-language letters: a dotted letter O, an accented U, and an accented O. Figure 8.11 illustrates the character codes used in the game, and Figure 8.12 is an example of the game being played.

A smiley is lost when he is caught by a chasing monster. There are four smileys allowed per game. The number of smileys left to play are displayed below the maze number at the top left of the screen. The variable PL in the program determines how many smileys are allowed per maze—this can be easily changed to other values.

The smiley face is moved about by pressing a cursor control key for the desired direction of movement. As it moves over dots, the score is incremented. If it moves into a monster or a monster moves into it, the smiley is lost. If all the dots in a maze are eaten, then a new maze is displayed, the game continues anew, and you are awarded an additonal two smileys for the next round.

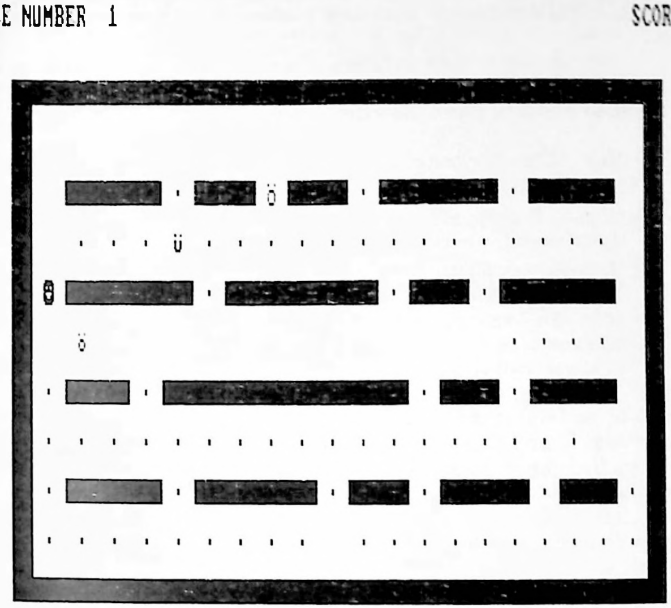
Each maze is randomly created to add some further variety to the game. The mazes consist of five rows of dots, where a dot is represented by the ASCII character code 249, interspersed between four rows of block characters, which are character codes 219. Each block character row may have from three to six openings. An overall border of block characters surrounding the nine rows completes the maze.

Movement of objects in a maze is essentially like all other animation. The basic difference is that additional checks must be made to ensure that objects are moved to a valid spot. We must check for the maze walls and the various objects that might be in the maze paths: dots, monsters, and smiley (the player).

MAZE SYMBOL	CHARACTER CODE
☺	2
ö	148
Ö	153
ü	154
·	249

FIGURE 8.11. Maze Chase game character codes.

FIGURE 8.12. Maze Chase game example.



Controlling the movement of the monsters is complicated somewhat by the need to give some degree of intelligence to the direction in which they move. This might range from a strictly random movement to having the monster always move in the direction of the player. In our game, we will let each monster have an increasing degree of intelligence which will control its movement toward the player. Monster 1 moves in a strictly random fashion; monster 2 moves toward the player only 25 percent of the time; and monster 3 moves in the direction of the player 75 percent of the time.

Once a monster starts moving, changing its direction need only be done when the monster encounters an intersection. At this point, the choice for movement is either to continue as before (no change), to turn up, or to turn down. Choosing to turn will then be based on a random choice or in a direction toward the player, depending on

which monster it is. To check for an intersection requires the use of the SCREEN statement for each new position that a monster moves to.

Overall control of the monsters is maintained with variables in five "M" arrays. These arrays keep track of the following for each of the three monsters where the index i may be 1, 2, or 3:

- MR(i), MC(i)—Row and column of monster i.
- MRI(i), MCI(i)—Monster's row and column increments.
- MD(i)—A direction code from 0 to 3 indicating the direction this monster is currently moving.
- MA(i)—Score value at which time monster i is activated in the game. This allows each monster to appear in sequence.
- ME(i)—The erased character code for this monster. It keeps track of the character code this monster erased when it was last moved. This is usually a blank space or dot in the maze row.

The variables R and C are used for smiley's current position; the variables RI and CI are used for the current direction increments. Recall that this is similar to the programming used in the other character-mode animation games. Each time smiley changes position, it is erased by printing a blank space, and is then reprinted at its new position. If smiley moves over a dot, the score is updated and a check is made to see if this was the last dot in the game. If it is, the player is awarded two smileys, a new maze is created, and the game continues. If this dot was not the last dot in the game, smiley just keeps on moving about eating dots until it hits a wall or monster, Program 8.6 is the complete Maze Chase game.

```
0 ' MAZE  Maze chase
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 ' Evade the monsters that chase you through a randomly created maze.
50 '
60 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
70 DEFINT A-Z : RANDOMIZE VAL(RIGHT$(TIME$,2))
80 PRINT"MAZE  -  Maze Chase Game" : PRINT
90 PRINT "Press <SPACE> to begin play": WHILE INKEY$<>" ": WEND
100 MN=0 : TS=0 : PL=4  ' maze number, total score, smileys
110 '
120 ' create random maze
130 '
140 CLS : TD=0 : MN=MN+1 : SC=0 : DOT$=" "+CHR$(249)+" "
150 FOR MR=1 TO 9
```

```
160 D=0 : L$=DOT$
170 IF (MR MOD 2)=1 THEN 240
180    FOR I=1 TO 5
190    IF I=3 OR RND>.25 AND ASC(L$)<>32 THEN L$=DOT$+L$ : D=D+1 ' openings
200    L$=STRING$(3*INT(2+RND*3),219)+L$
210    NEXT
220 L$=DOT$+L$ : D=D+2
230 IF LEN(L$)<>60 THEN 160 ELSE 260
240 L$="" : D=D+20
250    FOR I=1 TO 20 : L$=L$+DOT$ : NEXT ' row of dots
260 LOCATE MR*2+4,7 : PRINT L$
270 LOCATE MR*2+5,7 : PRINT SPACE$(60)
280 TD=TD+D
290 NEXT MR
300 COLOR 2 ' green borders
310 LOCATE 4,5 : PRINT STRING$(64,219); ' top
320 FOR R=5 TO 23
330 LOCATE R,5 : PRINT STRING$(2,219); ' left side
340 LOCATE R,67: PRINT STRING$(2,219); ' right side
350 NEXT
360 LOCATE 24,5 : PRINT STRING$(64,219); ' bottom
370 COLOR 7 ' white interior
380 LOCATE 1,1 : PRINT "MAZE NUMBER ";MN;TAB(66);"SCORE";TS
390 ON KEY(11) GOSUB 1220 : ON KEY(12) GOSUB 1230
400 ON KEY(13) GOSUB 1240 : ON KEY(14) GOSUB 1250
410 '
420 ' init variables for this maze
430 '
440 FOR I=1 TO PL-1 : LOCATE 2,3*I : PRINT CHR$(2); : NEXT ' smileys
450 TD=TD-2 : K=0 ' total dots (less 2 for entrances)
460 R=6 : C=35 : RI=0 : CI=0 ' smiley's position & increments
470 MX(1)=148: MX(2)=153: MX(3)=154  ' monster char codes (3 only)
480 FOR I=1 TO 3
490 MR(I)=22 : MC(I)=35 ' initial row,column of monster
500 ME(I)=32 : MA(I)=(I-1)*15 ' erased code, when activated
510 MRI(I)=0 : MCI(I)=3*SGN(RND-.5) ' row,column increments
520 MD(I)=-1 : IF MCI(I)>0 THEN MD(I)=1 ' direction code
530 NEXT
540 FOR K=11 TO 14 : KEY (K) ON : NEXT ' cursor key traps on
550 LOCATE R,C,0 : PRINT CHR$(2);
560 '
570 ' main loop: process and move smiley
580 '
590 RN=R+RI : CN=C+CI : P=SCREEN(RN,CN) ' set new position
600 IF P=219 THEN 690        ' hit wall?
610 LOCATE R,C : PRINT " "; ' erase old position
620 IF P>32 AND P<219 THEN GOSUB 1340 : GOTO 690 ' hit monster?
630 R=RN : C=CN                    ' save new position
640 LOCATE R,C : PRINT CHR$(2); ' move smiley
650 IF P=249 THEN GOSUB 1290     ' ate a dot?
660 '
670 ' move monsters using M*(I) arrays
680 '
690 FOR I=1 TO 3 : MP=0
700 IF SC<MA(I) THEN 1040
710 RM=MR(I) : CM=MC(I) : MRI=MRI(I) : MCI=MCI(I) ' monster position & incr.
720 P=SCREEN(RM+2,CM) : IF P<128 OR P>219 THEN MP=MP+1
730 P=SCREEN(RM-2,CM) : IF P<128 OR P>219 THEN MP=MP+1
```

```
740 P=SCREEN(RM,CM+3) : IF P<128 OR P>219 THEN MP=MP+1
750 P=SCREEN(RM,CM-3) : IF P<128 OR P>219 THEN MP=MP+1
760 IF MP>2 THEN 780   ' at an intersection
770 GOSUB 1470 : IF P<128 OR P>219 THEN 910      ' continue same direction?
780 IF RND<.5 THEN 860                           ' move randomly?
790 UD=0 : LR=0 : IF RND>.5 THEN 830             ' NO: chase player
800 UD=1 : MRI=2*SGN(R-RM) : MCI=0 : GOSUB 1470 '  either by row
810 IF P<128 OR P>219 THEN 910                   '  if possible
820 IF LR THEN 860
830 LR=1 : MRI=0 : MCI=3*SGN(C-CM) : GOSUB 1470 '  or by column
840 IF P<128 OR P>219 THEN 910                   '  if possible
850 IF NOT UD THEN 800
860 ND=RND*4 : NT=1                              ' YES: random direction
870 IF NT<4 AND (ND MOD 2)=(MD(I) MOD 2) THEN 900  ' no opposites
880 ON ND+1 GOSUB 1400,1410,1420,1430            ' set increments
890 GOSUB 1470 : IF P<128 OR P>219 THEN 910      ' move here OK?
900 IF NT<4 THEN ND=(ND+1) MOD 4 : NT=NT+1 : GOTO 870 ELSE 1040
910 '
920 ' direction set, now move monster I
930 '
940 LOCATE RM,CM : PRINT CHR$(ME(I)); ' reset last position
950 LOCATE RN,CN : PRINT CHR$(MX(I)); ' display at new
960 MR(I)=RN : MC(I)=CN : MCI(I)=MCI : MRI(I)=MRI ' save it
970 IF MD(I)=ND THEN 1020
980 IF MRI<0 THEN MD(I)=0 : GOTO 1020
990 IF MCI>0 THEN MD(I)=1 : GOTO 1020
1000 IF MRI>0 THEN MD(I)=2 : GOTO 1020
1010 MD(I)=3
1020 IF P=2 THEN GOSUB 1340 ' hit smiley?
1030 IF P<>2 THEN ME(I)=P ELSE ME(I)=32
1040 NEXT : GOTO 590 ' continue main loop
1050 '
1060 ' start another maze
1070 '
1080 FOR D=1 TO 99 : C$=INKEY$ : NEXT
1090 LOCATE 2,20 : PRINT "Maze ";MN;"completed. Press <SPACE> for next one"
1100 WHILE INKEY$<>" ": WEND: PL=PL+2: GOTO 140
1110 '
1120 '  end of game
1130 '
1140 LOCATE 2,20 : PRINT "GAME OVER.  Want to play some more?";
1150 FOR D=1 TO 99 : C$=INKEY$ : NEXT
1160 LOCATE 3,15: PRINT "Press Esc to exit or <SPACE> for a new game...";
1170 C$="" : WHILE C$="" : C$=INKEY$ : WEND
1180 IF ASC(C$)=27 THEN 9050 ELSE IF C$=" " THEN 100 ELSE 1170
1190 '
1200 ' player's cursor key traps
1210 '
1220 RI=-2 :CI=0  :RETURN 'up
1230 RI=0  :CI=-3 :RETURN 'left
1240 RI=0  :CI=+3 :RETURN 'right
1250 RI=+2 :CI=0  :RETURN 'down
1260 '
1270 ' score
1280 '
1290 TS=TS+1 : SC=SC+1 : LOCATE 1,71 : PRINT TS;
1300 IF SC<TD THEN RETURN ELSE RETURN 1080
1310 '
1320 ' smiley hit
```

```
1330 '
1340 PL=PL-1 : IF PL=0 THEN RETURN 1140 ' game over
1350 LOCATE 2,3*PL : PRINT " "; 'erase 1 more smiley
1360 R=6 : C=35 : RI=0 : CI=0 : RETURN
1370 '
1380 ' monster direction changes
1390 '
1400 MRI=-2:MCI=0 :RETURN 'up
1410 MRI=0 :MCI=+3:RETURN 'right
1420 MRI=+2:MCI=0 :RETURN 'down
1430 MRI=0 :MCI=-3:RETURN 'left
1440 '
1450 ' check monster's new move
1460 '
1470 RN=RM+MRI : CN=CM+MCI : P=SCREEN(RN,CN) : RETURN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 8.6 MAZE - Maze chase game

Variations

There are many possible variations for the games in this chapter.

1. In the Bust 'em game, you can put partial walls in the middle of the playing area. This will give the ball some additional places from which to rebound.

2. Additional separate barriers to break through can also be easily added. By spacing the barriers apart, the ball will also bounce about more. Blank spaces in the barriers instead of a solid barrier is another possibility.

3. Levels of playing difficulty can be incorporated in Bust 'em by adjusting the delay factor after the ball is moved. Let this factor be based on user input at the beginning of the game.

4. In the two Lunar Lander games, the number of levels can be increased. Be careful to keep the landing pads from getting so small that the LEM cannot possibly land on them.

5. Keep track of how many times the cursor keys are pressed as a measure of the "fuel" that is used by your LEM. A timer could also be used to add further difficulty to the game.

6. The Satellite Invasion game could have additional satellites or projectiles. This is a major change and would require some of the techniques used in the Asteroid Attack program for keeping track of everything.

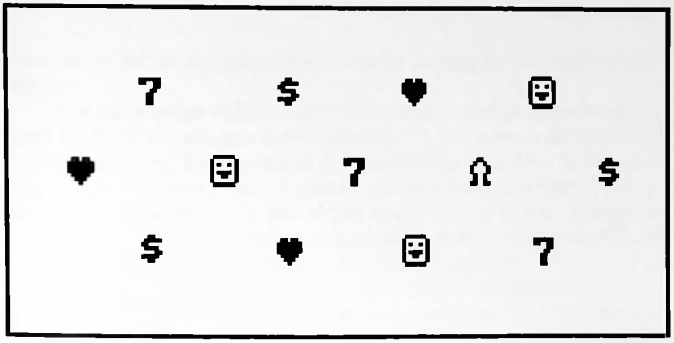
7. Asteroid Attack could be enhanced with a more complex background of planets and larger stars. Change the program to a medium-resolution screen mode and add color to the objects.

8. Let the player move the rocket diagonally instead of vertically or horizontally.

9. The laster beam could be handled to fire in the direction that the player last moved. This would allow firing it upward, downward, or behind the rocket. Note that a laser beam image array would be required for each of the four directions. The checks to see if an asteroid is hit would also depend on the direction of the beam.

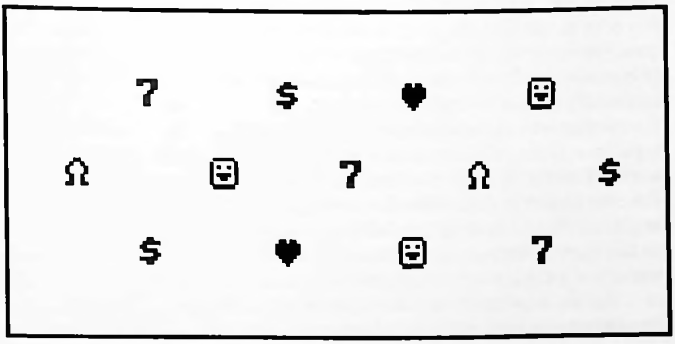
10. The Maze Chase game can have additional monsters by adding the appropriate codes to the M* arrays and changing the limits in the FOR / NEXT loop that processes the monsters.

11. The inclusion of "power pills" and the player's attacking the monsters is another possibility. (The latter is a fairly involved change.)



9

Educational Games



One of the greatest potential benefits of computerization is in the area of education. It is generally felt that the dust has yet to settle on achieving this potential. Although a number of general forms for computer-aided education are beginning to take shape, a generally accepted term for what to call "educational software" has not solidified as of yet. Gameware, courseware, CAI (computer-aided instruction), and CAE are a few of the terms often used.

TYPES OF EDUCATIONAL GAMES

Aside from simple computer games, which teach a certain degree of "computer literacy" or familiarity with computing equipment, two general classes of educational software can be considered: drill and practice courseware, and tutorial courseware. The distinction is in the overall goal of each type of courseware. Drill and practice is generally oriented to reenforcing some basic skill or some fixed body of knowledge. This leads to two subclasses: repetitive skill tasks and knowledge-based (facts) assessment.

Arithmetic problem solving is the most common example of the repetitive skills drill and practice courseware. The user is given a steady stream of problems that he must solve. It is understood that the user knows how to solve the problems. The emphasis is on affirming that skill by tracking how many problems are answered correctly in the group, timing the user's solution, providing him with a wide variety of problems to solve, and so on.

Another example of this might be a spelling bee. The computer displays an image and part of the spelling of the image. The user is

then required to complete the word by keying in the appropriate letters.

The knowledge or fact-based drill and practice programs have a fixed body of knowledge that is provided to the user in a number of forms. Learning the alphabet by associating common words that begin with the letter of some picture on the screen is a typical example. Another example is the states and capitals game. A state is pictured on the screen, and the name of the state and its capital must be identified. In each example, a fixed set of facts are "taught" to the user: the letters of the alphabet, or the states and their capitals.

Tutorial courseware can also be considered in two broadly based forms: electronic flashcard or textbook information, and interactive experimentation. The first form takes a body of knowledge, facts, concepts, and procedures and presents them to the user as a series of "screens" which the user "steps" through. A screen may present one or more facts and/or questions for the user.

If questions are asked, then the next screen is based on the reply given. Thus, if a question is correctly answered, the next set of facts is presented. If a question is answered incorrectly, a series of additional screens might be used to point out the student's errors, allowing the student to review the earlier screens and then answer the questions again. The degree of interaction in these systems is fairly limited: yes/no questions, true/false answers, and multiple choice.

The limiting factor in the "flashcard" approach is the limitation in user replies for progress assessment. In particular, the user will not be answering any context based or free form type of questions. Consider some courseware that "teaches" BASIC programming. The part of the course that considers the coding of loops could not simply ask the user to write a loop to find the largest number in an array. Any number of valid answers is possible using the FOR/NEXT and WHILE/WEND loop statements, or the IF/THEN and GOTO statements. We have a way to go before courseware will be designed that can utilize this type of interaction.

Another approach is interactive experimentation. The body of knowledge is presented in such a way that the user can examine and review the effects of what is being taught instead of using a strict question/answer approach. The computer takes on the role of a laboratory or even a lab partner. Consider a physics course. After a few concepts on gravity are introduced, the user might be allowed to experiment with throwing an object of a fixed size at some angle, and

with some variable amount of force. The computer then dynamically animates the results so students can see what they have done in terms of the distance or height achieved. Variations in the experiment could allow changing the mass of the object, the trajectory, and the force. How about changing gravity (as in the case of being on the moon or in a space shuttle)? Needless to say, the possibilities are open ended.

Note that the computer can also play the role of a "partner" in the above scenario. If some desired results are not achieved after a few experiments, then the computer could "suggest" some changes that the students should make in their experiment.

In general, courseware is dominated by drill and practice. The programming and overall development overhead is somewhat less for this type of courseware, as opposed to interactive experimentation, which can easily become very involved. Other forms of courseware will soon evolve over time as the developers continue their efforts. The field of artificial intelligence will have an important impact on future courseware designs.

In this chapter, we will provide two types of drill and practice programs: Arithmetic Drills, and States and Capitals.

Arithmetic Drills

The basic idea in the Arithmetic Drills program is to reenforce the skills of simple addition, subtraction, multiplication, and division. The program first allows selection of one of the four operators. (See Figure 9.1 for an illustration of the operator selection screen in the program.) After the operator is chosen, this screen is cleared and randomly created problems are presented, as shown in the illustration of an addition problem in Figure 9.2.

Problems are displayed by using oversized digits and operator symbols. Underlines are displayed for the digits in the part of the problem that is missing, which may be either of the problem's two operands or its answer. Your answer is keyed in and the program indicates whether you got it right or wrong with a yes or a no. Another problem is then presented.

A new operator can be selected by pressing the Esc key when the program requests you to press the space key for the next problem. The main menu is then displayed again, and another operator

FIGURE 9.1. Operator selection in the Arithmetic Drills program.

```
ADRILLS - Arithmetic Drills

        Select an operator to use:
     1) Addition
     2) Subtraction
     3) Multiplication
     4) Division
     5) Exit program
       Your choice (1 to 5)?
```

FIGURE 9.2. Example problem in the Arithmetic Drills program.

```
What is the answer? 9
```

may be chosen or you may exit from the program. The program is listed as Program 9.1.

Two key points about the program should be considered. First, the character set for the problems is coded as a series of subroutines. The subroutines locate to the row and column as set by the variables R and C, where they print the appropriate digit, operator, or the underline symbol (for digits missing in the problem).

```
0 ' ADRILLS   Arithmetic Drills
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 ' Practice answering basic arithmetic problems
50 '
60 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
70 DEFINT A-Z : RANDOMIZE VAL(RIGHT$(TIME$,2))
80 '
90 ' display menu and get type of problem
100 '
110 CLS : PRINT "ADRILLS - Arithmetic Drills" : PRINT
120 PRINT : PRINT TAB(10);"Select an operator to use:"
130 PRINT : PRINT TAB(8);"1) Addition"
140 PRINT : PRINT TAB(8);"2) Subtraction"
150 PRINT : PRINT TAB(8);"3) Multiplication"
160 PRINT : PRINT TAB(8);"4) Division"
170 PRINT : PRINT TAB(8);"5) Exit program"
180 PRINT : PRINT TAB(10);"Your choice (1 to 5)? "; : LOCATE ,,1
190 OP=0 : WHILE OP<1 OR OP>5 : OP=VAL(INKEY$) : WEND
200 IF OP=5 THEN C$=CHR$(27) : GOTO 9050 ' exit?
210 '
220 ' choose problem values, and display them
230 '
240 CLS : C=1 : R=5                ' initial problem row and column
250 ON OP GOSUB 450,460,470,480    ' set random values for problem
260 WN=INT(RND*3)                  ' pick value thats not displayed
270 N=N1 : IF WN=0 THEN A=N1 : GOSUB 650 ELSE GOSUB 490  ' 1st number, N1
280 N=OP : GOSUB 580    '  the operator,
290 N=N2 : IF WN=1 THEN A=N2 : GOSUB 650 ELSE GOSUB 490  ' 2nd number, N2
300 N=5  : GOSUB 580    '  the equals sign.
310 N=NA : IF WN=2 THEN A=NA : GOSUB 650 ELSE GOSUB 490  ' answer, NA
320 '
330 ' process user's answer
340 '
350 LOCATE 18,12:LINE INPUT "What is the answer? ",N$ : M=VAL(N$)
360 N=A  : C=AC : GOSUB 490    ' fill in the actual answer
370 IF M=A THEN LOCATE 20,12:PRINT "Your answer is correct";:GOTO 390
380 LOCATE 20,8:PRINT "Sorry. You missed that one. The answer is";A;
390 LOCATE 24,1: PRINT"Press <SPACE> for next problem, or Esc for a different dr
ill";
400 C$="" : WHILE C$="" : C$=INKEY$ : WEND
410 IF C$=CHR$(27) THEN 110 ELSE IF C$=" " THEN 240 ELSE BEEP : GOTO 400
420 '
430 ' set problem numbers and answer by operator type
440 '
450 N1=RND*12+1 : N2=RND*12+1 : NA=N1+N2 : RETURN    ' addition
460 N1=RND*12+1 : N2=RND*N1    : NA=N1-N2 : RETURN    ' subtraction
470 N1=RND*12+1 : N2=RND*12+1 : NA=N1*N2 : RETURN    ' multiplication
480 N2=RND*12+1 : NA=RND*12+1 : N1=N2*NA : RETURN    ' division
490 '
500 ' display the digits, D, in the number N
510 '
520 N$=STR$(N) ' convert N to a string
530 FOR I=2 TO LEN(N$) ' get each digit
540 D=VAL(MID$(N$,I,1))
550 ON D+1 GOSUB 1000,1100,1200,1300,1400,1500,1600,1700,1800,1900
560 C=C+9  ' next column
570 NEXT : RETURN
580 '
```

```
590 ' display operators, based on N=1 to 5 for + - * / =
600 '
610 ON N GOSUB 2040,2100,2140,2210,2270 : C=C+9 : RETURN
620 '
630 ' display an underline for each digit in the number N
640 '
650 N$=STR$(N) : AC=C  ' save column of number to be guessed
660 FOR I=2 TO LEN(N$)
670 GOSUB 2000 : C=C+9 ' display underline
680 NEXT : RETURN
690 '
700 ' oversized characters
710 '
1000 LOCATE R,C: PRINT " *****"
1010 LOCATE ,C : PRINT "*     *"
1020 LOCATE ,C : PRINT "*     *"
1030 LOCATE ,C : PRINT "*     *"
1040 LOCATE ,C : PRINT "*     *"
1050 LOCATE ,C : PRINT "*     *"
1060 LOCATE ,C : PRINT " ***** "
1070 RETURN
1080 '
1090 '
1100 LOCATE R,C: PRINT "   *"
1110 LOCATE ,C : PRINT "  **"
1120 LOCATE ,C : PRINT "   *"
1130 LOCATE ,C : PRINT "   *"
1140 LOCATE ,C : PRINT "   *"
1150 LOCATE ,C : PRINT "   *"
1160 LOCATE ,C : PRINT "  ***  "
1170 RETURN
1180 '
1190 '
1200 LOCATE R,C: PRINT "  ***"
1210 LOCATE ,C : PRINT " *   *"
1220 LOCATE ,C : PRINT "*     *"
1230 LOCATE ,C : PRINT "    *"
1240 LOCATE ,C : PRINT "  *"
1250 LOCATE ,C : PRINT " *"
1260 LOCATE ,C : PRINT "*******"
1270 RETURN
1280 '
1290 '
1300 LOCATE R,C: PRINT " *****"
1310 LOCATE ,C : PRINT "*     *"
1320 LOCATE ,C : PRINT "      *"
1330 LOCATE ,C : PRINT "  ****"
1340 LOCATE ,C : PRINT "      *"
1350 LOCATE ,C : PRINT "*     *"
1360 LOCATE ,C : PRINT " ***** "
1370 RETURN
1380 '
1390 '
1400 LOCATE R,C: PRINT "   **"
1410 LOCATE ,C : PRINT "  * *"
1420 LOCATE ,C : PRINT " *  *"
1430 LOCATE ,C : PRINT "*   *"
1440 LOCATE ,C : PRINT "*******"
1450 LOCATE ,C : PRINT "    *"
1460 LOCATE ,C : PRINT "    *  "
```

```
1470 RETURN
1480 '
1490 '
1500 LOCATE R,C: PRINT "*******"
1510 LOCATE ,C : PRINT "*      *"
1520 LOCATE ,C : PRINT "*"
1530 LOCATE ,C : PRINT "*******"
1540 LOCATE ,C : PRINT "      *"
1550 LOCATE ,C : PRINT "*      *"
1560 LOCATE ,C : PRINT " ***** "
1570 RETURN
1580 '
1590 '
1600 LOCATE R,C: PRINT " *****"
1610 LOCATE ,C : PRINT "*      *"
1620 LOCATE ,C : PRINT "*"
1630 LOCATE ,C : PRINT "* ****"
1640 LOCATE ,C : PRINT "**     *"
1650 LOCATE ,C : PRINT "*      *"
1660 LOCATE ,C : PRINT " ***** "
1670 RETURN
1680 '
1690 '
1700 LOCATE R,C: PRINT "*******"
1710 LOCATE ,C : PRINT "*      *"
1720 LOCATE ,C : PRINT "      *"
1730 LOCATE ,C : PRINT "     *"
1740 LOCATE ,C : PRINT "    *"
1750 LOCATE ,C : PRINT "   *"
1760 LOCATE ,C : PRINT "  *     "
1770 RETURN
1780 '
1790 '
1800 LOCATE R,C: PRINT " *****"
1810 LOCATE ,C : PRINT "*      *"
1820 LOCATE ,C : PRINT "*      *"
1830 LOCATE ,C : PRINT " ******"
1840 LOCATE ,C : PRINT "*      *"
1850 LOCATE ,C : PRINT "*      *"
1860 LOCATE ,C : PRINT " ***** "
1870 RETURN
1880 '
1890 '
1900 LOCATE R,C: PRINT " *****"
1910 LOCATE ,C : PRINT "*      *"
1920 LOCATE ,C : PRINT "*      *"
1930 LOCATE ,C : PRINT " ******"
1940 LOCATE ,C : PRINT "      *"
1950 LOCATE ,C : PRINT "*      *"
1960 LOCATE ,C : PRINT " ***** "
1970 RETURN
1980 '
1990 '
2000 LOCATE R+6,C : PRINT STRING$(7,220) ' underline
2010 RETURN
2020 '
2030 '
2040 LOCATE R+2,C : PRINT "   ";CHR$(220) ' plus + addition
```

```
2050 LOCATE R+3,C : PRINT " ";STRING$(2,220);CHR$(219);STRING$(2,220)
2060 LOCATE R+4,C : PRINT "   ";CHR$(219)
2070 RETURN
2080 '
2090 '
2100 LOCATE R+3,C : PRINT " ";STRING$(5,220) ' minus - subtraction
2110 RETURN
2120 '
2130 '
2140 LOCATE R+1,C : PRINT " \   /"  ' "times" x multiplication
2150 LOCATE R+2,C : PRINT "  \ /"
2160 LOCATE R+3,C : PRINT "  / \"
2170 LOCATE R+4,C : PRINT " /   \"
2180 RETURN
2190 '
2200 '
2210 LOCATE R+2,C : PRINT "   ";CHR$(220) ' division sign
2220 LOCATE R+3,C : PRINT " ";STRING$(5,220)
2230 LOCATE R+4,C : PRINT "   ";CHR$(220)
2240 RETURN
2250 '
2260 '
2270 LOCATE R+2,C : PRINT " ";STRING$(5,220) ' equals sign
2280 LOCATE R+4,C : PRINT " ";STRING$(5,220)
2290 RETURN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program 9.1 ADRILLS - Arithmetic drills

The ten digits are comprised of groups of asterisks formed in a 7-by 7-character box. This is done to make the overall format of the digits more easy to discern as you key in the program. A more "solid" image might be created by using the business graphics symbols; in particular, the block characters, code numbers 219 through 223. When the business graphic codes are listed in the program, though, it would be difficult to determine which codes were used and their positions in the program lines.

Each digit's subroutine consists of seven LOCATEs and PRINTs. They start at line 1000, in the program for digit 0, through line 1900, for the digit 9. To print a single digit, the row and column variables R and C are set and the statement shown in program segment 9.1 is executed, where D is the digit being printed.

The operators use other symbols. They are formed primarily with the block character code number 220. It is used to form the

underline symbol, the horizontal line in the plus sign, the minus sign, and the top line in the equal sign. The vertical line in the plus sign consists of the character code 219. The multiplication sign consists of the slash, /, and backslash, \, to form the diagonals of the X. The division sign uses a horizontal dash with dots above and below the dash, formed by the character code 249. The bottom dash of the equal sign is the character code 223.

The STRING$ and CHR$ function are used to create the character codes in each symbol so you can easily key the character codes when copying the program. The display of these symbols is done in the same manner as the digits. A GOSUB is made to the appropriate subroutine, which LOCATES to the appropriate row and column, and PRINTs the symbol.

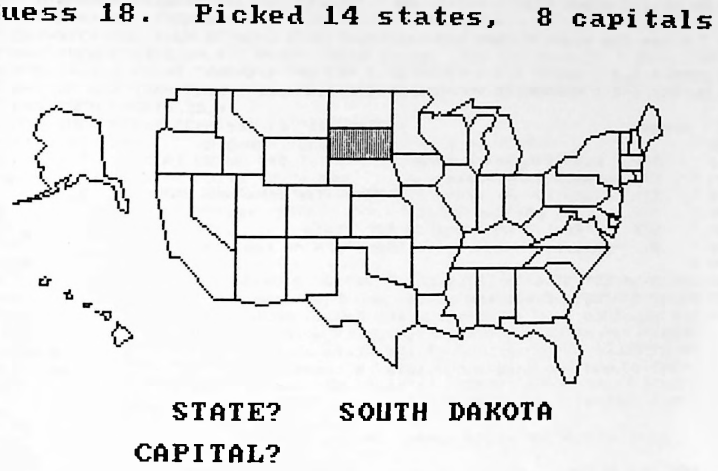
A second noteworthy point concerns the numbers used in the problems. The values for the problems, both the operands and their answers, are randomly chosen in such a way as to ensure that they are one- or two-digit values in the range of 1 to 12. The exception is for multiplication answers and dividends for division problems, which may be three digits. Four subroutines, one for each operator type, have been coded to select these values. Note that the subroutines also make sure that there are no negative-valued subtractions or division-by-zero problems.

The variables N1 and N2 represent the first and second operands in each problem. The answer is stored in variable NA. A fourth variable, WN, is set to 0, 1, or 2 to indicate which value in the problem to leave blank. It is checked in the program prior to displaying each number in the problem. Thus, if WN is 1, then the second operand is displayed as a series of underlines. When the dashes are printed, the column for this number and its actual value are stored in AC and NG. They are used after the user makes his or her entry to display the actual value on top of the underlines.

States and Capitals

The States and Capitals game displays a map of the continental USA on the screen. A state is highlighted, and the user is asked to guess the state's name and its capital. (See Figure 9.3.) The answers are keyed in at the bottom of the screen in the blank areas provided.

FIGURE 9.3. States and Capitals game example.



After both replies are made, the computer responds with an "atta-boy" or a "sorry . . .", along with the correct state and/or capital name. A running tally is kept of how many guesses have been made, and how many states and capitals were guessed correctly. Entering the letter X for either a state or a capital guess ends the program.

To run this game on an IBM PC requires a color graphics adapter, which is standard on the PCjr. This is necessary because the map is drawn in medium-resolution graphics, SCREEN 1. Such a map in the character-mode graphics would not allow the detail to show all fifty states.

The overall map consists of fifty individual DRAW statement strings which outline each state. These strings and other data are listed in the DATA statement table at the end of the program. It includes the state's name, its capital, the position on the screen where the state is to be drawn for the overall map, an interior position point in the state for coloring (or highlighting) it, and the DRAW statement string for the overall state. Program 9.2 is the complete listing of the game.

```
0 ' STATES  States and Capitals Game
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 ' Guess the state's name and capital
50 '
60 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
70 DEFINT A-Z : RANDOMIZE VAL(RIGHT$(TIME$,2))
80 '
90 ' Arrays:
100 '
110 '    STN - complete state name
120 '    STC - capital of state
130 '    STL - location of state for Move (for complete map)
140 '    STP - PAINT point position
150 '    STD - actual DRAW commands for state
160 '    G   - selection table (so there are no repeats)
170 '
180 DIM STN$(50),STC$(50),STL$(50),STP$(50),STD$(50),G(50)
190 PRINT"STATES - Guess the states and their capitals":PRINT
200 DEF SEG=&H40 : IF (PEEK(&H10) AND &H30)<>&H30 THEN 220 ' correct adapter?
210 PRINT"Program requires color/graphics adapter." : GOTO 9030
220 PRINT"Enter 'X' to either of the state or capital questions"
230 PRINT"to end the game and display a summary of your guesses."
240 PRINT: PRINT"Press <SPACE> to begin the game."
250 WHILE INKEY$<>" " : WEND : CLS
260 '
270 '  load arrays for state names, capitals and DRAW statements
280 '
290 FOR I=1 TO 50
300 READ STN$(I),STC$(I),STL$(I),STP$(I),STD$(I) ' load arrays
310 NEXT
320 '
330 '  Set mode, palette, DRAW map, set-up screen etc.
340 '
350 CLS:SCREEN 1,0:COLOR 0,0:DRAW "S4; BM24,0
360 FOR I=1 TO 50
370 DRAW "C3 BM72,32 Xstl$(i); Xstd$(i);"  ' DRAW state outlines
380 DRAW "BM72,32 Xstp$(i);" : PAINT STEP(+0,+0),2,3  '  PAINT them
390 NEXT
400 LOCATE 1,1:PRINT"Guess      Picked    states,    capitals";
410 LOCATE 20,11:PRINT"STATE?"; : LOCATE 22,9:PRINT"CAPITAL?";
420 G=0 : GS=0 : GC=0  ' guess counts
430 '
440 '  begin game
450 '
460 RS=INT(RND*50)+1  ' randomly select a state
470 FOR I=1 TO G : IF G(I)=RS THEN 460 ELSE NEXT       ' check if already used
480 DRAW "BM72,32 Xstp$(rs);" : PAINT STEP(+0,+0),1,3  '  highlight it
490 '
500 '  call prompt processor for state guess
510 '
520 ROW=20:COL=20:GOSUB 960:ST$=A$
530 IF A$="X" THEN 880
540 '
550 '  and the capital
560 '
570 ROW=22:COL=20:GOSUB 960:CA$=A$
580 IF A$="X" THEN 880
590 '
```

```
600 '  process the guesses
610 '
620 A=0  ' answer flag
630 IF ST$=STN$(RS) THEN GS=GS+1 ELSE A=1   ' state correct?
640 IF CA$=STC$(RS) THEN GC=GC+1 ELSE A=A+2 ' the capital?
650 IF A=0 THEN LOCATE 24,7 : PRINT "Good going!  You got them."; : GOTO 700
660 LOCATE 24,3 : PRINT "Sorry!! ";
670 IF A=1 OR A=3 THEN PRINT "State is ";STN$(RS);
680 IF A=3 THEN LOCATE 25,9
690 IF A>1 THEN PRINT "Capital is ";STC$(RS);
700 G=G+1 : G(G)=RS   ' up guess count and save state number
710 LOCATE 1,7  : PRINT USING "##."; G;  ' display count,
720 LOCATE 1,19 : PRINT USING "##"; GS;  ' how many states guessed,
730 LOCATE 1,30 : PRINT USING "##"; GC;  ' and how many capitals
740 LOCATE 3,8  : PRINT "Press <SPACE> for next state";
750 WHILE INKEY$<>" " : WEND
760 '
770 ' clear parts of screen for next guess
780 '
790 FOR R=24 TO 25 : LOCATE R,1 : PRINT SPC(39); : NEXT ' answers,
800 LOCATE 20,20:PRINT SPC(18);  ' state reply area,
810 LOCATE 22,20:PRINT SPC(18);  ' captial reply area,
820 LOCATE 3,4 : PRINT SPC(34);  ' continue message,
830 DRAW "BM72,32 Xstp$(rs);"  : PAINT STEP(+0,+0),2,3 ' and the state
840 IF C$<>"X" AND C$<>"x" AND G<50 THEN 460
850 '
860 ' end of game wrap-up
870 '
880 FOR R=20 TO 25 : LOCATE R,1 : PRINT SPC(39); : NEXT
890 LOCATE 20,9 : PRINT "Game over.  You picked";
900 LOCATE 21,7 : PRINT GS;"states, and";GC;"capitals";
910 LOCATE 22,9 : PRINT "out of";G;"guesses.";
920 GOTO 9030 ' bypass subroutine
930 '
940 '  get guesses
950 '
960 LOCATE ROW,COL : PRINT SPC(18); ' clear answer area
970 LOCATE ROW,COL : LINE INPUT A$ ' get answer
980 IF A$="" THEN BEEP : GOTO 960  ' null not allowed
990 FOR I=1 TO LEN(A$)
1000 C=ASC(MID$(A$,I,1)) : IF C=32 THEN 1030
1010 IF C>96 AND C<123 THEN C=C-32 : MID$(A$,I,1)=CHR$(C) 'lower to upper case
1020 IF C<64 OR C>91 THEN BEEP : GOTO 960 ' alphabetic characters only
1030 NEXT
1040 RETURN
1050 '
1060 ' State names, positions, etc.
1070 '
1080 DATA ALABAMA,MONTGOMERY, "BM+140,+68" , "BM+141,+69" , "R17 D19 L7 D4 M-5,-
1 L5 U22"
1090 DATA ALASKA,JUNEAU, "BM-48,+10" , "BM-44,+16
1100 DATA "M+6,-3M+4,+2M+10,+2M+4,+2M+2,+20M+7,+2M+1,+6M+2,+1D5M-4,-2M-2,-8M-6,-
2L4M-2,-2L2M-1,+2M-5,+3U2M+2,-3M-6,+3M+1,+3M-8,+3M-10,+2M+7,-2M+8,-4M-4,-4M+1,-4
L2M-1,-4M+4,-4M-5,-4M+3,-8"
1110 DATA ARIZONA,PHOENIX, "BM+34,+56" , "BM+35,+57" , "R22 D24 M-10,+2 M-12,-2
U24"
1120 DATA ARKANSAS,LITTLE ROCK,"BM+111,+60" , "BM+112,+61" , "R22 M-5,+8 M-3,+10
 L12 U2 M-2,-1 U15"
1130 DATA CALIFORNIA,SACRAMENTO, "BM+0,+32" , "BM+1,+33" , "R15 D16 M+19,+14 D18
 L9 M-15,-16 L4 M-6,-24 U8"
```

```
1140 DATA COLORADO,DENVER, "BM+56,+36" , "BM+57,+37" , "R28 D20 L28 U20"
1150 DATA CONNECTICUT,HARTFORD, "BM+205,+30" , "BM+206,+31" , "R6 M+1,+4 M-7,+1
U5"
1160 DATA DELAWARE,DOVER, "BM+200,+47" , "BM+201,+49" , "M+5,+3 M-1,+2 L4 U5"
1170 DATA FLORIDA,TAMPA, "BM+150,+87" , "BM+151,+88" , "R7 D2 R17 U2 R6 M+7,+13
M-3,+12 L4 M-5,-8 U6 M-10,-6 M-5,+3 M-10,-4 U4"
1180 DATA GEORGIA,ATLANTA, "BM+157,+68" , "BM+158,+69" , "R13 M+12,+14 M-2,+5 L6
 D2 L17 U21"
1190 DATA HAWAII,HONOLULU, "BM-48,+72" , "BM-14,+94"
1200 DATA "BM+0,+1 M+2,-1M+2,+1M-1,+2L3U2 BM+10,+5R1M+1,-1M+1,+3L2M-1,-2 BM+8,+4
R3M-1,+1L2U1 BM+7,+4M+1,-1R3D2M-2,+1M-1,-2M-3,-1 BM+7,+8M+2,-2 U2M+5,+4M+1,+2M-3
,+1M-2,+2M-2,-1M-1,-4"
1210 DATA IDAHO,BOISE, "BM+25,+0" , "BM+26,+1" , "R7 D8 M+14,+10 D14 L18 U14 L3
U18"
1220 DATA ILLINOIS,SPRINGFIELD, "BM+125,+28" , "BM+126,+29" , "R11 M+4,+4 D21 M-
6,+1 M-13,-18 M+4,-4 U4"
1230 DATA INDIANA,INDIANAPOLIS, "BM+140,+32" , "BM+141,+33" , "R12 D17 M-10,+5 M
-2,-1 U21"
1240 DATA IOWA,DES MOINES, "BM+102,+24" , "BM+103,+25" , "R21 M+2,+4 D4 M-4,+4 L
15 M-4,-10 U2"
1250 DATA KANSAS,TOPEKA, "BM+84,+40" , "BM+85,+41" , "R22 M+5,+4 D12 L27 U16"
1260 DATA KENTUCKY,FRANKFORT, "BM+134,+54" , "BM+136,+56" , "M+6,-1 M+2,+1 M+10,
-5 M+8,+3 M+7,+6 M-5,+2 L29 M+1,-6"
1270 DATA LOUISIANA,BATON ROUGE,"BM+113,+78" , "BM+114,+79" , "R12 M+2,+4 M-3,+4
 R6 M+3,+9 M-20,-6 U11"
1280 DATA MAINE,AUGUSTA, "BM+218,+4" , "BM+219,+6" , "M+4,+2 M+2,+10 M-5,+2 M-3,
+6 L1 M-2,-10 M+5,-10"
1290 DATA MARYLAND,ANNAPOLIS, "BM+178,+44" , "BM+179,+45" , "R16 M+6,+3 D5 R4 M-
2,+4 M-8,-4 M-1,-4 M-3,+2 M-5,-1 U3 L7 U2"
1300 DATA MASSACHUSETTS,BOSTON, "BM+205,+24" , "BM+206,+25" , "R11 M-1,+6 L10 U6
"
1310 DATA MICHIGAN,LANSING, "BM+125,+9" , "BM+136,+9" , "M+13,-6 D3 R12 M-1,+2 M
+1,+1 R4 M+3,+9 M+3,+6 M-3,+8 L17 M+5,-8 M-3,-6 M+3,-4 M+1,+1 M+2,-5 M-1,-1 L7 M
-4,+5 M-3,-5 L8"
1320 DATA MINNESOTA,ST PAUL, "BM+102,+0" , "BM+103,+1" , "R15 M+3,+3 M-6,+5 D8 M
+9,+8 L21 U24"
1330 DATA MISSISSIPPI,JACKSON, "BM+128,+68" , "BM+129,+69" , "R12 D22 M-8,+2 M-2
,-6 L6 M+3,-4 M-2,-4 M+3,-10"
1340 DATA MISSOURI,JEFFERSON CITY,"BM+106,+36" , "BM+107,+37" , "R15 M+13,+18 M-
1,+6 L22 U16 M-5,-4 U4"
1350 DATA MONTANA,HELENA, "BM+32,+0" , "BM+33,+1" , "R42 D18 L28 M-14,-10 U8"
1360 DATA NEBRASKA,LINCOLN,"BM+74,+26" , "BM+75,+27" , "R28 M+4,+10 D4 L22 U4 L1
0 U10"
1370 DATA NEVADA,CARSON CITY, " BM+15,+32" , "BM+16,+33" , "R19 D30 M-19,-14 U16
"
1380 DATA NEW HAMPSHIRE,CONCORD, "BM+210,+14" , "BM+211,+15" , "R3 M+2,+10 L6 M+
1,-10"
1390 DATA NEW JERSEY,TRENTON, "BM+200,+36" , "BM+201,+38" , "M+6,+2 M-1,+10 M-5,
-1 M-6,-3 M+6,-4 U4"
1400 DATA NEW MEXICO,SANTA FE, "BM+56,+56" , "BM+57,+57" , "R22 D22 L13 D2 L9 U2
4"
1410 DATA NEW YORK,ALBANY, "BM+176,+28" , "BM+178,+28" , "M+4,-3 R5 M+5,-7 M+10,
-3 R5 D20 M+1,+3 M-6,-2 M-5,-1 U3 L19 U4"
1420 DATA NORTH CAROLINA,RALEIGH, "BM+170,+60" , "BM+172,+62" , "R30 M-12,+12 M-
6,-6 L10 M-2,+2 L10 M+10,-8"
1430 DATA NORTH DAKOTA,BISMARK, "BM+74,+0" , "BM+75,+1" , "R28 D14 L28 U14"
1440 DATA OHIO,SPRINGFIELD, "BM+152,+32" , "BM+153,+33" , "R18 D12 M-10,+8 M-8,-
3 U17"
```

```
1450 DATA OKLAHOMA,OKLAHOMA CITY, "BM+78,+56" , "BM+79,+57" , "R33 D19 M-20,-5 U
10 L13 U4"
1460 DATA OREGON,PORTLAND, "BM+4,+14" , "BM+4,+16" , "M+3,+2 R18 D2 R3 D14 L28 U
16 M+4,-2
1470 DATA PENNSYLVANIA,HARRISBURG, "BM+170,+32" , "BM+171,+32" , "M+6,-4 D4 R19
D3 M+5,+1 D4 M-6,+4 L24 U12"
1480 DATA RHODE ISLAND,PROVIDENCE, "BM+211,+30" , "BM+212,+31" , "R4 D4 L3 M-1,-
4"
1490 DATA SOUTH CAROLINA,COLUMBIA, "BM+172,+66" , "BM+174,+68" , "R10 M+6,+6 M-6
,+10 M-12,-14 M+2,-2"
1500 DATA SOUTH DAKOTA,PIERRE, "BM+74,+14" , "BM+75,+15" , "R28 D12 L28 U12"
1510 DATA TENNESSEE,NASHVILLE, "BM+133,+60" , "BM+135,+62" , "R37 M-10,+8 L32 M+
5,-8"
1520 DATA TEXAS,AUSTIN, "BM+78,+60" , "BM+79,+61" , "R13 D10 M+22,+6 D13 M-9,+5
D12 M-9,-6 M-8,-12 L7 D2 L5 M-10,-10 U2 R13 U18"
1530 DATA UTAH,SALT LAKE CITY, "BM+34,+32" , "BM+35,+33" , "R12 D4 R10 D20 L22 U
24"
1540 DATA VERMONT,MONTPELIER , "BM+205,+15" , "BM+206,+16" , "R5  M-1,+9 L4 U9"
1550 DATA VIRGINIA , RICHMOND , "BM+162,+60" , "BM+168,+59" , "M+5,-2 M+13,-5 M+
2,-5 M+3,+1 M+5,+1 M+3,-2 M+1,+4 M+6,+8 L38"
1560 DATA WASHINGTON,OLYMPIA, "BM+8,+0" , "BM+9,+1" , "R17 D16 L18 M-3,-2 U10 R4
 U4"
1570 DATA WEST VIRGINIA,CHARLESTON, "BM+170,+44" , "BM+172,+46" , "R8 D2 R7 D3 M
-3,-1 M-2,+5 M-13,+5 M-7,-6 M+10,-8"
1580 DATA WISCONSIN,MADISON, "BM+114,+8" , "BM+115,+9" , "R6 D2 M+3,-2 M+2,+1 R8
 M+3,+5 D14 L11 M-2,-4 M-9,-8 U8"
1590 DATA WYOMING,CHEYENNE, "BM+46,+18" , "BM+47,+19" , "R28 D18 L28 U18"
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 LOCATE 25,1:PRINT SPC(39);
9040 LOCATE 25,1:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

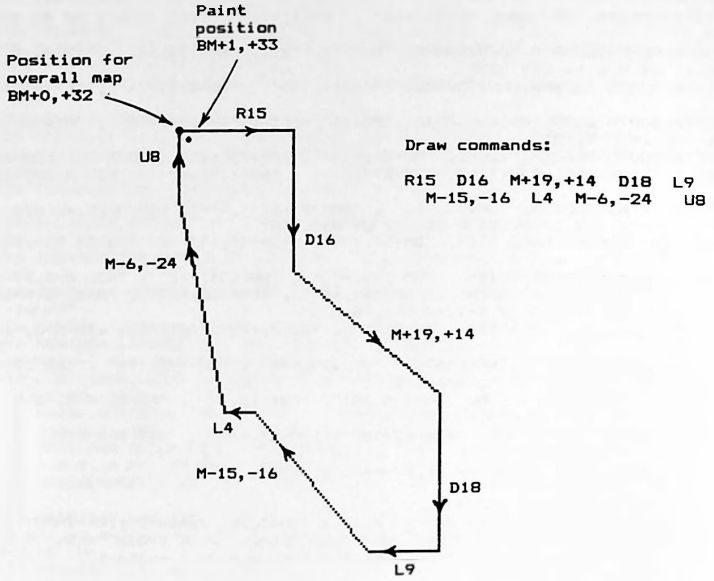
Program 9.2 STATES - States and capitals game

The following string arrays are used in the program for these tables:

- STN$—The name of the state.
- STC$—The capital of the state.
- STL$—Draw command for the location of the state in the overall map.
- STP$—Draw command for the point to begin painting the state.
- STD$—Commands to draw the outline of this state.

Figure 9.4 illustrates the DRAW statement commands for the state of California. In order to keep the map relatively simple, liberties were taken in creating the outlines of some of the states. Many states have

FIGURE 9.4. Individual map example for the state of California.



a fairly simple form (such as Colorado, which requires only four commands, one for each side). Alaska, on the other hand, is much more complex and thus requires far more DRAW commands.

The two position points are used as follows. The first one is that point on the screen where the individual state is to be drawn when the entire map of the USA is displayed. The DRAW strings for each state are relative to point (0,0). Thus, they can be used in a group (as done here) or individually. This could be used in a different game where, say, one state is drawn at a time and the user must identify individual cities within the state.

Such a display could be preceded by different positioning points and a change in the scale factor to allow for a larger display than that used in this game (which uses a scale of 4). See Figure 9.5 for an expanded map of California with the neighboring states of Nevada, Utah, and Arizona. The large dot in each state defines the starting

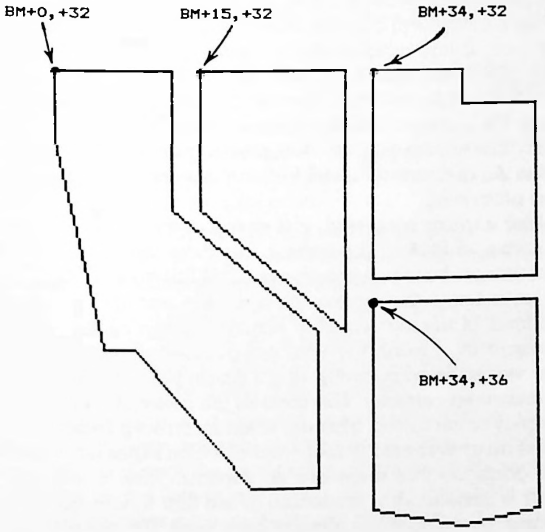
point for drawing each state. This starting point is initially established by the DRAW command BM in the array STL$. It moves to a point on the screen but does not draw a line there.

The other position point just defines a point inside each state for painting purposes. After locating to this point with the BM command, a PAINT statement is executed to color that state's interior. A different color is used to highlight the state when it's time to guess the state's name and capital. The DRAW commands for the position string are stored in the array STP$.

The game uses colors based on the medium resolution color group palette number 0. The background color is black, number 0. Outlines for the state are drawn in color 3, yellow, which is also the color of the text in the game. The states in the map are colored red, number 2, and are highlighted with color 1, green, when the guesses are made.

For those of you with a black-and-white monitor, the highlighting will be hard to distinguish from the initial state's color (colors 1

FIGURE 9.5. Four-state map example.



and 2 will appear the same). Therefore, the initial state color should be changed to 0, for a black background. This is used in two places: when the map is initially drawn and when the state to be guessed is erased from the screen.

The overall sequence in displaying the entire map is as follows:

1. Move to the point on the screen for each state in the overall map.
2. DRAW the outline of the individual state.
3. Move to some point within this state.
4. PAINT the interior of the state.

This is done for each entry (state) in the DATA statement table.

Guesses are selected by randomly choosing a number between 1 and 50. The number is used to select the state to be guessed, which is then highlighted in color 1. The user's guesses are entered and compared to the correct name of the state and its capital.

A subroutine at line 960 accepts the user's guesses. It disallows any null entries from being entered and converts all letters to upper case in the event that the shift key has not been used. This is required because the state and capital's names in the program's tables are in upper case. If any nonalphabetic characters are entered, the subroutine BEEPs and asks the guess again. Note the use of the ROW and COL variables to define the row and column where each guess is made. They are set to 20,20 when the state's name is to be guessed, and to 22,20 for the capital. Acceptable guesses are stored in the variable A$ and are returned to the main part of the program for further processing.

After a guess is entered, it is stored in the variable ST$ for the state name, or SC$ for the capital. They are then compared to the correct values, based on the arrays STN$(RS) and STC$(RS), where RS is an index to the random state to be guessed. The variable A keeps track of the four possible results: 0 when neither guess was made correctly, 1 if only the state was guessed correctly, 2 if only the capital was guessed correctly, and 3 if both the state and the capital were answered correctly. This controls the message that follows and the display of the correct answers when incorrect guesses are made.

The array G is used to keep track of which states have been used in the game so that there are no repeats. After a random state number is generated, it is checked to see that it does not appear in this array, meaning it has already been used. If it has already been used, another random state number is chosen.

The variables G, GS, and GC are the "guess counts." They keep track of how many guesses have been made, how many states were guessed correctly, and how many correct capitals were guessed, respectively. They are displayed after each guess at the top of the screen to show how well you are doing. They are also printed at the end of the game to summarize the final results of play.

Variations

Listed below are some possible enhancements that can be made to the games. It should not be difficult to derive many more.

1. Convert Arithmetic Drills to a graphics mode. Each digit and the operator symbols could be displayed using DRAW commands, as in the States and Capital game.

2. Allow a user of the program to narrow the range of values selected for the problems. For instance, in multiplication problems, a single operand might be specified instead of any number (as is now the case).

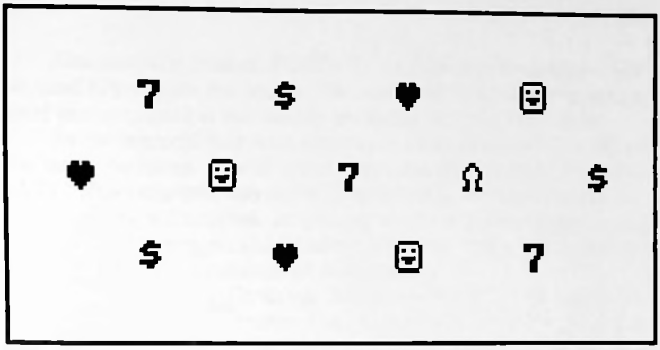
3. The problems could be oriented vertically. This would probably be best coordinated with a graphics-mode version of the program.

4. A tally of how well the user does might be appropriate. It would keep track of how many problems were attempted and how many were correctly answered. A timer might also keep track of how long it took the user to answer the problems.

5. The States and Capital game could be modified to allow the user to guess just the state or just the capital. Other cities could be added, such as the largest city in each state.

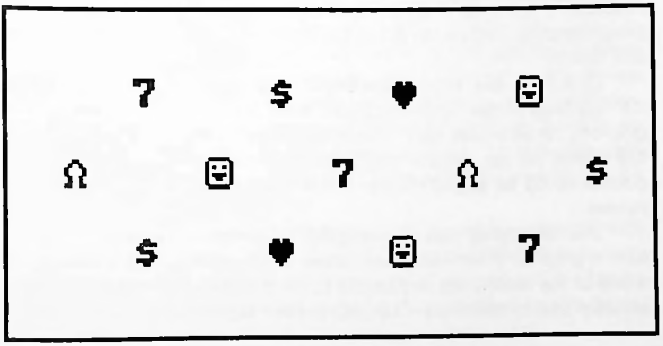
6. Just an outline of the USA could be drawn when displaying the overall map. Thus, only the state to be guessed would be highlighted, and outlines for its neighboring states would not be seen. This will add some more difficulty to the game.

7. For both programs, a "prompt processor" could be used for accepting the problem answers and the state or capital guesses. This processor would accept the user's entry character by character and change case as the letters are keyed. It could also filter out any extraneous keystrokes such as the cursor control keys, Esc, Ins, Del, and so forth.



10

Wrap-Up



In this chapter, we will tie together some loose ends and mention a few other topics related to our discussion of computer gaming.

PROGRAMMING LANGUAGES

The main theme of this book has been using your IBM PC and PCjr and the BASIC programming language to write computer games. BASIC is an obvious choice for a number of reasons: It's easily learned and used; the language takes good advantage of the graphics capabilities in the machine; and every PC and PCjr comes with some form of BASIC.

Others might say that BASIC is a poor choice: It's slow; it's not standardized so that programs cannot be easily moved from one machine to another; it often requires using cryptic PEEK and POKE programming, and so on. All of this may be true, but how did it affect our results?

Of course, the interpretive BASIC is slow, but the tradeoff is that it's very easy to use, write programs with, and debug programs in. In general, its slowness only affects the action graphics games. All of the others do not require any major speedup, although the board games could be slightly faster while the computer is pondering its moves.

The programs can be compiled to obtain faster speed in the action games. When compiled, some delay will have to be incorporated in the animation sequences since the speedup will be approximately five to ten times their interpretive equivalents.

Alternatively, Pascal, FORTH, C, or assembly language could be used to program the games. Unfortunately, much of the graphics used in our games is not readily available in these languages.

As for compatibility with computers other than the PC or PCjr, the issue becomes one of using machines that provide Microsoft BASIC. The programs use no PEEK and POKE statements or assembly-language subroutines, so as long as the machine supports this form of BASIC, you should be okay. However, this is not a blanket endorsement and guarantee of compatibility.

Indeed, the new Cartridge BASIC on the PCjr has seven different graphics screen modes (the PC has only two). The games in this book do not take advantage of all of the Cartridge BASIC features so they will be compatible with the earlier PC and Compiler BASIC. Modifying many of the programs for these additional modes should be fairly easy. For example, the Asteroid Attack game in Chapter 8 could be programmed in four colors instead of its present black and white.

Writing Your Own Games

One of the biggest chores in creating a game is designing and coding the graphics displays. Two choices that might be considered are to use some sort of purchased screen-generator software that allows the design of formatted screens which are stored in disk files until they are displayed again. A BLOAD statement, for instance, can load the screen file directly into the display buffer memory of your monitor.

When designing character mode displays, you can simply enter the text to be displayed (less multiple colors) directly on the screen using BASIC's internal line editor. As long as you do not press Enter, the screen will retain the characters entered for viewing. This allows you to experiment with different screen layouts. For the extended ASCII characters, you must type their numeric codes in with the Alt key and numeric keypad.

Once an acceptable screen is created, it must be recoded with the usual sequence of LOCATE, COLOR, and PRINT statements. Still, this approach enables you to see what is being coded better than writing it on graph paper or some video display screen form.

For the graphics modes, the use of purchased software to design the screen may be the easiest choice. On the other hand, as you work with them, coding the proper LINE, PSET/PRESET, and DRAW statements to create the desired image becomes fairly easy to do. Additional files for your screens are not needed, and they are easily edited within your program. Ultimately, the choice of techniques depends on the complexity of the images being designed.

Dreaming up an idea for a game is another matter altogether. For starters, the many variations provided at the end of each chapter provide some stepping-off points for additional games. There are also some references in Appendix D that mention sources for other computer games and ideas.

A key ingredient here is imagination and some luck. Many a game that is seemingly based on a good and fun idea may just not work out. Game play, graphical representation, the flow of the game, the goal, the degree of difficulty, its "learnability," and many other factors come into play. Determining the best mixture of these items for a particular game is a difficult task.

Tying Together the Programs

A diskette of all the game programs in this book can be purchased from the author. See the enclosed coupon. If you have bought the diskette, then you also have a menu program along with the game programs. This menu lists each game that is available and allows you to call the games up for play. Once a game is finished, it will return to the menu to allow the selection of another game or ending of your session on the computer.

You can also call each game program directly, such as RUN "STATES for the States and Capitals game. If started directly, the program will return to BASIC when you end the game. This allows you to "tinker" with individual programs without having them always go back to the menu.

Those of you who bought only the book and would like the benefits of such a menu program should look at Appendix C, A Menu Program. In general, the program maintains a list of the game names and the program filename to call. Once a game is selected, a CHAIN statement calls it up. To return to the menu, a CHAIN back to MENU is performed. You can easily incorporate any number of

games in the program by modifying the DATA statement table accordingly.

In the appendices, there are a number of other topics related to the rest of the book which some readers might find of interest. This information is not essential to the overall understanding of the previous chapters. The subject of random number generation is the most relevant.

I hope that you enjoy all of the games as much as I have enjoyed programming and playing them.

APPENDIX A: RANDOM NUMBERS

Randomness in one form or another is common to most computer games. It is inherently required to produce some element of surprise or variation in the game; otherwise, predictable patterns of play would result and your game would become more of an exercise in memory than in skill or reflexes.

The concept of randomness can easily become quite complex. There are some very heavy mathematics associated with describing the extent of true randomness and whether or not it prevails; this will be covered only briefly here. Numerous other texts and articles can be found that explain these concepts further. Our purpose here is to introduce what is needed to support the use of randomness in the games developed in the book and to offer some ideas for additional development.

Random Number Generation

BASIC's answer to providing randomness is the RND function. This function is best looked upon as a random number generator, or as simply a random number. Each time it is used, it will take on a new randomly determined value that is between zero and one. For example, the statement:

```
X = RND
```

causes the variable X to take on some arbitrary value in the range from zero to one. (Note that this assumes that X is defined as either a

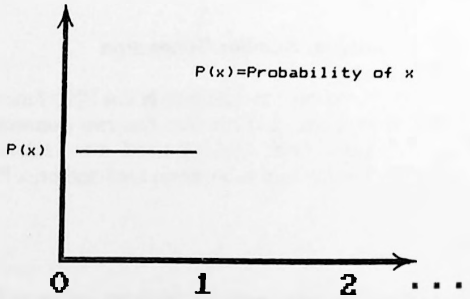
single- or double-precision number. If it is not, that is, if it is an integer, then BASIC will round X to the nearest integer, and a 0 or 1, depending on whether RND is less than or greater than 0.5, will result.) Alternate forms of RND using an argument will be discussed at a later point.

The particular values that X takes on are termed as following the uniform distribution. This is due to the fact that no one value returned by RND is any more or less likely to be produced than any other value. Hence, the numbers are uniformly distributed over the entire range, which is from zero to one.

If one was to plot a large sample of the values generated by RND, then a graph of the form depicted in Figure A.1 would be produced. The value p, for the probability of each random number, is represented as some constant, nonzero value in the graph. This is in keeping with the fact that all values are equally likely to be produced. In reality, though, the value of p approaches zero since there are an infinite number of real numbers between zero and one. The probability of any one particular value is actually zero.

Before moving on and putting RND to use, it should also be pointed out that the term "pseudo-random" is more applicable to our discussion than simply the term "random." True random numbers from a statistical standpoint result from some real-life event, experiment, or survey. The role of statistics is to put some semblance of order to this randomness and describe with accuracy and a measurable degree of confidence the characteristics associated with it.

FIGURE A.1. Graph of X = RND



On the other hand, random numbers generated by computers are based on a software subroutine buried in the interpreter or compiler or in some subroutine for the computer language being used. As such, the numbers they produce are very predictable and can often be reproduced at will whenever needed. This is important since the use of these numbers depends on how well the subroutine being used produces them. In general, the techniques utilized are based on well-founded procedures and usually provide a reasonably "random" set of numbers. The most widely utilized technique is called the *congruential method* and involves the use of modular arithmetic, as in BASIC's MOD operator. Further descriptions of these techniques can be found in the references.

The ability to reproduce a sequence of pseudo-random numbers is quite useful. For instance, programs can be tested with the same "random" values each time, or simulations repeated with the same numbers. To restart a sequence of numbers, or to change it, requires resetting what is known as the random function's *seed*. The seed is nothing more than a starting number from which the system's internal random number subroutine begins its random-number generation. Set the seed to the same value each time and the same set of numbers will be generated. (This was used by the MANOR program of Chapter 4 in creating the haunted house.)

To set the random number seed, you use the RANDOMIZE statement. Invoking RANDOMIZE without any number following the statement will cause the following prompt to be made at your terminal:

```
Random number seed (-32768 to 32767)?
```

You then enter some number in the range of −32768 to 32767 (which, by the way, is a 16-bit integer). The number you enter is then used as the basis for all random numbers subsequently generated by RND. Alternatively, you may code the RANDOMIZE statement as follows:

```
RANDOMIZE X
```

where X has the seed value you wish to use. This eliminates the prompt question, as described earlier.

A problem arises, though: How do you asign X some arbitrary (as in random) value to keep from using the same seed? Well, why not use the RND function? How about using something such as:

```
RANDOMIZE RND*1000
```

where the expression RND*1000 generates some random value that is now between 0 and 999, which becomes the new seed? Sorry, this won't do. Regardless of how you code the expression, the same sequence of random numbers is generated, since the same seed value will be used each time the program starts anew. It is surprising how often the code of this form can be found in programs, especially obtuse expressions using RND*RND/RND . . ., etc. They are all for naught.

A better choice is to use some seed value that is truly an arbitrary value that is independent of your program. One of the easiest sources of such values is to use the current time, such as the current seconds value or the number of seconds that has passed since the last hour began. Thus, an expression of the form

```
RANDOMIZE VAL(RIGHT$(TIME$,2))
```

can be used, which sets the seed to the current seconds value in the TIME$ variable. Thus, the seed value will be in the range of 0 to 59 for sixty possible values, or sixty possible random number sequences. In particular, this is the standard way of randomizing our games in this book. Each program includes in its start up the BASIC code as shown above or extended to:

```
RANDOMIZE VAL (MID$ (TIME $,4,2)) *60 + VAL (RIGHT$
(TIME$, 2))
```

This statement calculates seed values in the range of 0 to 3599. If the hour quantity is included, a modulo operator must be added to keep the resulting value, which is the number of seconds since midnight, from exceeding 32767.

```
RANDOMIZE ( VAL (RIGHT$ (TIME$,2)) *3600 + VAL (MID $
(TIME $,4,2) *60
+ VAL (RIGHT $(TIME$,2)) ) MOD 32767
```

Alternatively, some location in the operating system's memory can be used as a seed value. Obviously, finding some "randomly" changing memory location requires knowledge of the inner workings of your computer. For the PC, we can look through the BIOS listing in IBM's Technical Reference guide. A good possibility is the system's

internal clock, which is constantly updating two bytes in memory starting at hex segment address &H40 and offset &H6C. Thus, the following statement sets the seed to some value between 0 and 255:

```
DEF SEG=&H40 : RANDOMIZE PEEK(&H6C)
```

Under Version 2.0 of DOS, PC BASIC also allows using a new system variable called TIMER which can be used in much the same manner as the examples above. TIMER returns a value for the number of seconds since midnight for the current day. Note also that RANDOMIZE in Version 2.0 accepts any seed value and is not restricted to the range −32768 to +32767.

The random number seed can also be set by providing a negative argument to the RND function such as

```
X = RND(-13)
```

An argument of zero in RND causes it to repeat the last number generated. Thus, X and Y will be the same in the following:

```
X = RND : Y= RND(0)
```

Positive arguments serve only as dummy values and have no effect on the random numbers generated.

The random function is most often used in some expression that converts it to some other range of values. Before discussing the techniques used for this, we will close this section with a sample program that plots a bar graph of the RND function.

Program A.1 prints a horizontally oriented bar graph for a sample of RND-generated values. Ten bars are printed where each bar represents a range of values, as in 0 to 0.10, 0.10 to 0.20, . . . on up to 0.90 to 1.0. For a sample size of 100, you should expect approximately ten numbers per bar. It is interesting to note the variations from 10 that are actually generated. Change the seed value used and note the differences in the quantities of numbers produced for each range.

The program asks the user for the sample size and a seed value to use. The ten totals are tallied in the array S, one total for each range, as described earlier. The actual array index used and the totaling of each range is done with the expression:

```
R=RND : S(INT(R*10)) = S(INT(R*10)) + 1
```

```
0 ' RBAR  Random number bargraphs
10 MENU$="NO"
20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
30 '
40 '  Bargraph a set of random numbers.
50 '
60 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
70 DEFINT A-Z : DEFSNG R : DIM S(9)
80 PRINT "RBAR - Random number bargraphs" : PRINT
90 LINE INPUT "Enter sample size (10 or more, 0 exits program): ";N$
100 N=VAL(N$) : IF N=0 THEN C$=CHR$(27) : GOTO 9050 ELSE IF N>9 THEN 120
110 PRINT"Please enter a value of at least 10." : PRINT : GOTO 90
120 LINE INPUT"Enter RANDOMIZE seed value: ",N$ : RANDOMIZE VAL(N$)
130 '
140 ' tally random number values in array S
150 '
160 T=0 : FOR I=0 TO 9 : S(I)=0 : NEXT
170 FOR I=1 TO N
180 R=RND : T=T+R : S(INT(R*10))=S(INT(R*10))+1
190 NEXT
200 SMAX=0 : FOR I=0 TO 9 : IF S(I)>SMAX THEN SMAX=S(I)
210 NEXT I
220 '
230 ' bargraph heading
240 '
250 CLS:PRINT TAB(30);"RANDOM NUMBER TESTS"
260 PRINT TAB(20);"Sample size";N;"average is";T/N
270 RSCL=65/SMAX ' set scale for 65 column width bar
280 PRINT
290 '
300 ' print label, data, and bars
310 '
320 FOR I=0 TO 9
330 PRINT USING "< #.##  ####";(I+1)/10;S(I);
340 PRINT TAB(14);STRING$(S(I)*RSCL+1,220)
350 NEXT
360 '
370 ' print scale
380 '
390 PRINT TAB(14);"¦";
400 FOR I=1 TO 6:PRINT "---------¦";:NEXT
410 PRINT
420 FOR I=0 TO 6
430 PRINT TAB(I*10+13);    ' TAB to every 10th column, and
440 PRINT INT(I*10/RSCL); ' print the scale value for it
450 NEXT
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program A.1 RBAR - Random number bargraphs

where the function INT(R*10) calculates the index value. This expression saves the trouble of performing a series of IF statements to determine in which range the random number will fall.

Once the range totals are tallied, the maximum range is determined. It is then used as the basis for a scaling factor to plot the bars across sixty-five columns on the screen and also to calculate and print a set of scale values along the horizontal axis of the graph.

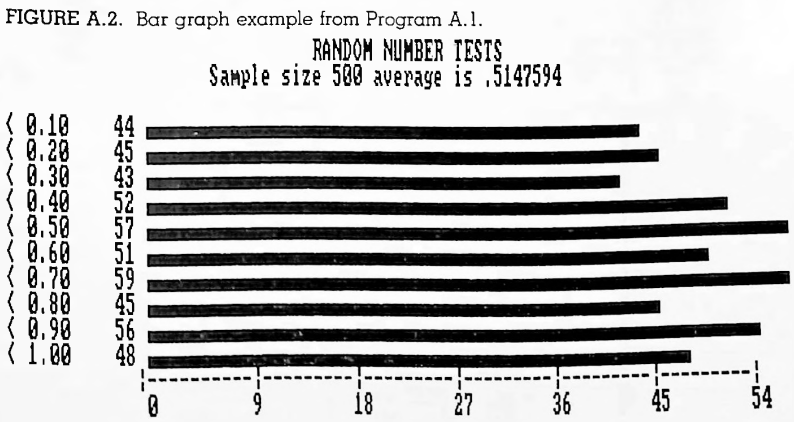
Figure A.2 is an example of a bar graph generated with a seed of 13 and a sample size of 500. Later in this appendix, a revised version of the program will be developed that may be used to analyze any range of random numbers, and is not limited to those values in the range of 0 to 1, as is Program A.1.

Types of Random Number Distributions

More often than not, the random number function is used in some expression that converts its value to some other range or form. For instance, to produce random numbers between 0 and 6 simply requires multiplying RND by 6, as in

```
X = RND*6
```

FIGURE A.2. Bar graph example from Program A.1.



And to change the range for 1 to 6, add one and multiply RND by five

```
X = 1 + RND*5
```

where RND*5 produces numbers in the range of 0 to 5, and adding one increases them to the desired range of 1 to 6. Figure A.3 is an illustration of the graph that represents the above expression.

In general, to generate random numbers in the range of A to B where B>A (A=1 and B=6 in the above example), the expression:

```
X = A + RND*(B-A)
```

can be used. If a negative-valued range is desired, such as −5 to +5, use the same expression. Don't forget that subtracting a negative number is the same as adding its positive equivalent. Thus,

```
X = -5 + RND*( +5 - (-5) )
```

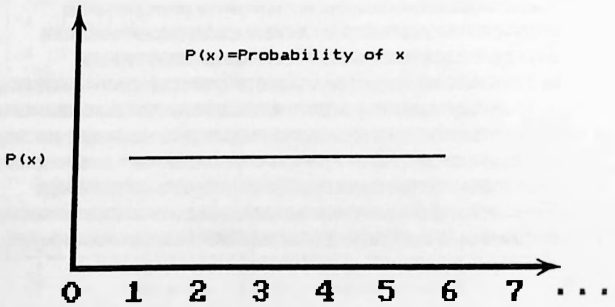
can be simplified to

```
X = -5 + RND*10
```

which produces the desired range of values between −5 and +5.

As mentioned earlier, these examples generate what is known as *continuous values*, since any number in the calculated range may be produced each time the expression is evaluated by BASIC.

FIGURE A.3. Graph of X = 1 + RND*5



For *discrete values*, or numbers which may take on values from a finite set of possibilities, expressions such as the one used in Program A.1 can be used. For example, to produce one of the numbers 0, 1, 2, 3, 4, or 5, the expression

```
X = INT(RND*6)
```

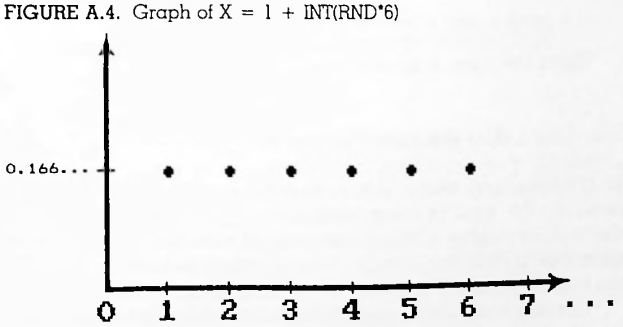
could be used. To generate a range of numbers with a nonzero lower limit is done by adding a constant to the expression. For integer numbers between 1 and 6, as in a dice simulation, we use the expression:

```
X = 1 + INT(RND*6)
```

A graph of the discrete-valued expressions shows a series of points, or dots, one for each possible value that may be returned. See Figure A.4, which is a graph of the above expression.

The term distribution has been used to describe the types of random numbers that certain formulas (expressions) can produce. So far, we have discussed the uniform distribution in both a discrete and continuous form. There are a number of other theoretical distributions, some of which can be useful in our gaming programs. We will describe a few of the more common ones in this section and show how they can be produced in BASIC.

FIGURE A.4. Graph of X = 1 + INT(RND*6)



The Normal Distribution

The normal distribution is often referred to as the "bell curve," and is frequently used in describing many real-life events. There are two parameters, such as the A and B values for the uniform distribution, which describe the normal distribution: They are the mean (or average) and the standard deviation. The mean is a sort of center point or the most likely value to be found from the normal distribution. The standard deviation is a measure of the degree of dispersion about the mean that the values may exhibit. Figure A.5 illustrates a normal curve with a mean or average of 50 and a standard deviation of 10.

There are a number of different ways of producing normal random variables. One technique is to simply add together twelve RND values and subtact six from the sum. This produces normal values that have a mean of zero and a standard deviation of one. To convert to some other mean and standard deviation, multiply this result by the desired standard deviation and add the desired mean. For example, the BASIC subroutine may be used where MEAN and STD.DEV describe the desired normal distribution.

```
900 REM generate Normalized random variables
910 X=0
920 FOR I=1 TO 12 : X=X+RND : NEXT
930 X = MEAN + (X-6!) * STD.DEV
940 RETURN
```

Another technique reduces the above summation to a single expression:

```
X = MEAN + SQRT(-2*LOG(RND)) * COS(6.283*RND) * STD.DEV
```

This latter form is known as the Box–Muller expression.

The Negative Exponential Distribution

The next time you wait in a line, consider the fact that, theoretically speaking, the time between people arriving to be served follows what is known as the negative exponential distribution. It has but one parameter to describe it, called *lamba*, which is both its mean and standard deviation. Figure A.6 is an illustration of this distribution.

The negative exponential disribution is often used in the simulation of waiting lines and their service facilities. The distribution

FIGURE A.5. Normal curve example.

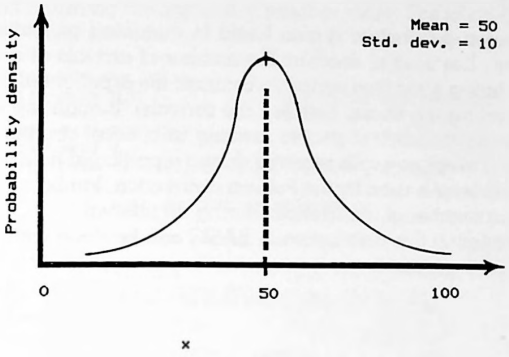
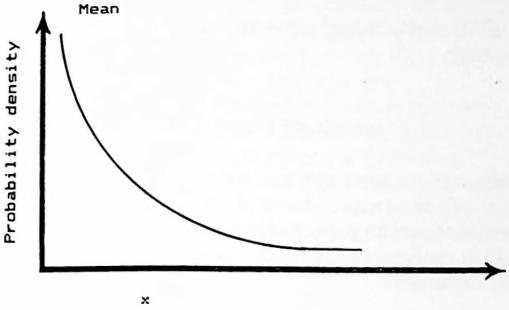


FIGURE A.6. Negative exponential distribution.



serves to describe the time between arrivals, such as customers at a ticket counter, patients at an emergency room, cars at a toll booth, and so on. This particular field of study is known as *queueing theory*, and has been applied to many such problems.

Producing negative exponential-distributed variables which have a mean and standard deviation equal to MEAN can be done with the following expression:

```
X = - MEAN * LOG( RND )
```

The Poisson Distribution

The Poisson distribution is also found in queueing or waiting-line problems. It is used to describe the number of arrivals at a service facility during some time period (in contrast, the exponential distribution describes the timing *between* the arrivals). It might be used to simulate the number of people arriving at a ticket counter or the number of telephone calls received during a particular time interval. One parameter is used by the Poisson distribution, lambda, which is the mean number of occurrences during an interval.

Simulating this distribution in BASIC can be done with the following code segment:

```
900 REM Poisson distributed variables
910 T=0
920 WHILE T<=1
930 T = T - LOG(RND) / MEAN
940 X = X + 1
950 WEND
960 RETURN
```

where MEAN is the mean value and X is the Poisson-distributed random value.

Empirical Distributions

It often occurs that instead of a well-defined theoretical distribution of random numbers, we need to produce numbers that follow somewhat of a "contrived frequency" distribution. For example, suppose it is desired to randomly produce the integers 1, 5, 7, and 8 with the following frequency:

Integer Value	Individual Frequency	Cumulative Frequence
1	40 percent	40 percent
5	30 percent	70 percent
7	20 percent	90 percent
8	10 percent	100 percent

The cumulative frequency column is simply a sum of the individual frequencies for the values preceding it. The total must be 100 percent; if it is not, then the individual frequencies are incomplete. With a table of the cumulative totals, generating random values is a matter

of using RND, searching through the table for the interval in which it lies, and returning the associated random value. The following code segment illustrates this:

```
100 FOR I=1 TO 4
110 READ V,F                ' get Value and Frequency
120 V(I)=V                  ' set value in array
130 C(I)=C(I-1)+F           ' set cumlative frequency
140 NEXT I
150 DATA 1, 0.40, 5, 0.30, 7, 0.20, 8, 0.1

     ... main portion of program ...

900 REM Discrete random value generator
910 R=RND
920 FOR I=1 TO 3            ' loop through table
930 IF R<=C(I) THEN 950     ' check this interval
940 NEXT                    ' continue to next one
950 X=V(I) : RETURN         ' return value based on I
```

The first portion of the program reads a DATA statement table containing the desired values and their frequency of occurrence. It then saves the values in array V and calculates the cumulative frequencies, each being a sum of the previous array elements and the current frequency. Note that this assumes a base of zero for the array C, since the subscript loop value begins at one.

Once the table is set, the subroutine need only loop through the first three values (or one less than the total number of values), since if R (the value for RND) is not less than the first three, then it must be by default in the last interval. When the FOR / NEXT loop is through, it leaves its index value at the STEP value one greater than its limit. Thus, I has a value of 1 to 4 at line 950, which is used to subscript the V array for the appropriate random value.

Other Distributions

Numerous other random-number distributions can be simulated: The Binomial distribution, the Chi-square, the Triangular, the Hypergeometric, the Geometric, the Lognormal, and others. The interested reader should refer to the references by Naylor and Watson for a summary of subroutines for these distributions and how they may be used. These references also contain extensive other references on the subject of random-number generation and producing random-number distributions.

Plotting Random Numbers

Program A.2 is a generalized version of Program A.1. The histogram is similar to a bar chart, where each bar in the chart represents an interval of values being plotted. The earlier program used constant values, ten intervals from zero to one. This limitation is relaxed in random-number histograms with the user of the program specifying the number of intervals, and the program calculating the appropriate interval values. The user also specifies the sample size and the seed value to use for the RND.

```
 0 ' RHISTO  Random number histograms
 10 MENU$="NO"
 20 ON KEY(1) GOSUB 9030 : KEY(1) ON ' "abort" key and MENU entry point
 30 '
 40 '  Graph random number distributions
 50 '
 60 SCREEN 0,1 : COLOR 7,0 : WIDTH 80 : KEY OFF : CLS
 70 DEFINT I,J,N :  DIM R(1000), DI(18), S(18), TD$(5)
 80 TD$(1)="Uniform" : TD$(2)="Normal" : TD$(3)="Negative Exponential"
 90 TD$(4)="Poisson" : TD$(5)="Exit from program"
 100 PRINT "RHISTO - Random number histograms" : PRINT
 110 PRINT : PRINT "Select a distribution:" : PRINT
 120 FOR I=1 TO 5 : PRINT TAB(7);I;"- ";TD$(I) : NEXT
 130 PRINT : PRINT "Your choice (1 to 5)?";
 140 WD=0 : LOCATE ,,1 : WHILE WD<1 OR WD>5 : WD=VAL(INKEY$) : WEND
 150 PRINT WD : IF WD=5 THEN C$=CHR$(27) : GOTO 9050
 160 LINE INPUT "Enter sample size (10 to 1000): ",N$: N=VAL(N$)
 170 IF N<10 OR N>1000 THEN PRINT "From 10 to 1000 only!" : GOTO 160
 180 LINE INPUT "Enter number of display intervals (2 to 18)? ",N$: IN=VAL(N$)
 190 IF IN<2 OR IN>18 THEN PRINT "From 2 to 18 only" : GOTO 180
 200 LINE INPUT "Enter random number seed value: ",N$: RANDOMIZE VAL(N$)
 210 PRINT : PRINT "Excuse me for a moment while I tally the random values.";
 220 '
 230 ' get random values based on type of distribution
 240 '
 250 T=0 : MIN=1E+17 : MAX=-MIN
 260 FOR I=1 TO N
 270 ON WD GOSUB 780,820,860,900 : R(I)=X : T=T+X
 280 IF X>MAX THEN MAX=X
 290 IF X<MIN THEN MIN=X
 300 NEXT
 310 '
 320 ' set display interval ranges
 330 '
 340 DEL=(MAX-MIN)/IN : DI(IN)=MAX
 350 FOR I=1 TO IN-1
 360 DI(I)=DEL*I+MIN
 370 NEXT
 380 '
 390 ' tally numbers by display interval
 400 '
 410 FOR I=1 TO N
 420 R=R(I) : FOR J=1 TO IN-1 : IF R>DI(J) THEN NEXT J
 430 S(J)=S(J)+1
```

```
440 NEXT I
450 SMAX=0 : FOR J=1 TO IN : IF S(J)>SMAX THEN SMAX=S(J)
460 NEXT
470 '
480 ' headings
490 '
500 CLS
510 H$="RANDOM NUMBER HISTOGRAMS - "+TD$(WD)+" Distribution"
520 PRINT TAB((80-LEN(H$))/2) ; H$
530 H$="Sample size"+STR$(N)+" mean value is"+STR$(T/N)
540 PRINT TAB((80-LEN(H$))/2) ; H$
550 S=65/SMAX ' set scale for 65 column width bar
560 PRINT
570 '
580 ' print label, data, and bar
590 '
600 FOR I=1 TO IN
610 PRINT USING "< ###.##  ###"; DI(I) ; S(I);
620 PRINT TAB(14);STRING$(S(I)*S+1,220)
630 NEXT
640 '
650 ' bottom scale
660 '
670 PRINT TAB(14);"¦";
680 FOR I=1 TO 6:PRINT "---------:";:NEXT
690 PRINT
700 FOR I=0 TO 6
710 PRINT TAB(I*10+13); ' TAB every 10th column
720 PRINT INT(I*10/S);  ' and print scale value
730 NEXT
740 GOTO 9000 ' bypass subroutines
750 '
760 ' Uniform dist., 0 to 1
770 '
780 X=RND : RETURN
790 '
800 ' Normal dist., mean 100, std.dev 20
810 '
820 X=100+20*SQR(-2*LOG(RND))*COS(6.283001*RND) : RETURN
830 '
840 ' Negative exponential, mean 10
850 '
860 X=-10*LOG(RND) : RETURN
870 '
880 ' Poisson dist., mean 20
890 '
900 S=0 : X=0
910 WHILE S<=1 : S=S-LOG(RND)/20 : X=X+1 : WEND : RETURN
9000 '
9010 ' exit to BASIC or menu, or restart program
9020 '
9030 COLOR 7,0:LOCATE 25,1:PRINT SPC(79);
9040 LOCATE 25,20:PRINT"Press Esc to exit or SPACE to restart";:C$=INPUT$(1)
9050 IF C$=CHR$(27) THEN IF MENU$="NO" THEN CLS:END ELSE CHAIN "MENU" ' exit
9060 IF C$=" " THEN IF MENU$="NO" THEN RUN ELSE RUN 20 ' restart
9070 GOTO 9040
```

Program A.2 RHISTO - Random number histograms

To illustrate the random-number distributions illustrated earlier, a set of subroutines is included in the program, which may be selected for use. To change a parameter associated with each distribution, such as its mean, the subroutine code must be modified.

The bar graphs in Figure A.7 illustrate the results of plotting some of the random distributions described above.

The program functions as follows. First, the desired distribution, the sample size, the number of intervals to plot (display), and the seed value are entered by the user. A FOR / NEXT loop then executes a GOSUB to the appropriate subroutine to calculate the random values that are stored in the array R. As the random values are calculated, the maximum value, minimum value, and their total are also determined.

After the sample is made, the minimum and maximum values are used along with the number of display intervals to determine the individual interval ranges. This is based on the minimum random value plus an increment, from one to the number of intervals, times

FIGURE A.7.1. Random-number histogram of normal distribution.

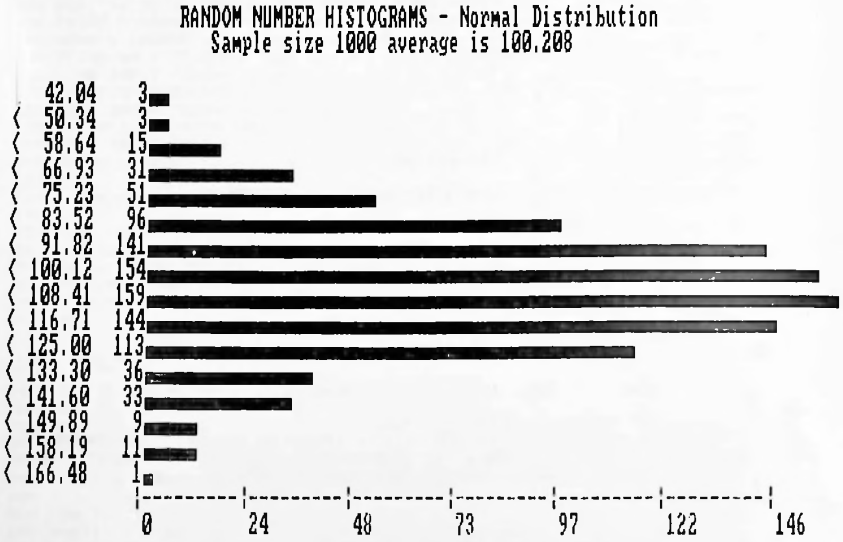
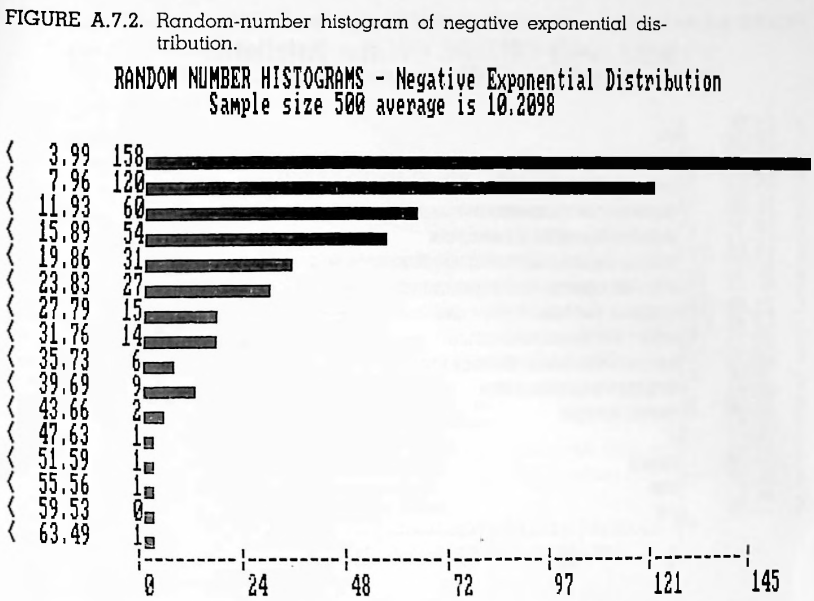


FIGURE A.7.2. Random-number histogram of negative exponential distribution.

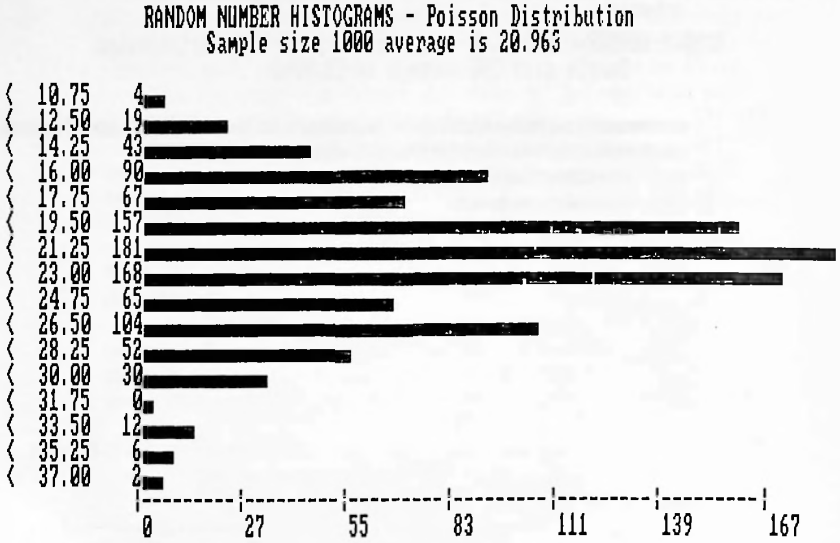


the size of each interval. The interval size is the range of random values (maximum less the minimum) divided by the number of intervals.

Once the display interval array DI is set, the random values in array R are examined in sequence to tally how many were generated per interval. The totals per interval are stored in the array S. After all values are tallied, the S array is plotted in a manner similar to the previous program. A vertical axis is printed based on the interval value DI, and the horizontal axis is created based on a scale factor which is also used to calculate the length of each bar that is displayed. The scale is determined by the maximum interval total in the same manner as the previous bar graph program.

Note that some of the distributions can take a few minutes to tally for large sample sizes. For instance, a sample of 2000 normal random variables will take about four minutes to plot. It is interesting to see how "smooth" the plots become as the sample size is increased.

FIGURE A.7.3. Random-number histogram of Poisson distribution.



APPENDIX B: USING THE SAMPLE-PROGRAMS DISKETTE

It can become quite tedious to type in the book's gaming programs; it is also error-prone and time-consuming. Therefore, the author offers a sample programs diskette which contains all the sample programs listed in this book. A menu program is also provided, which allows you to quickly find, load, and run any of the programs.

If you have bought only the book with the idea of just keying in a few of the samples, then that's okay. On the other hand, if you find yourself spending quite a bit of time keying in more and more of the programs, then the diskette should be purchased. It's a small price to pay for quite a lot of software and a savings in time.

In any event, to use the samples diskette requires a minimum of 64K of RAM, one double-sided disk drive, DOS, and BASICA or Cartridge BASIC for the PCjr. There are twenty-nine programs on the

disk. Twenty-eight of the programs have names as listed in the sample programs index, such as PCSAYS.BAS for the program to play the game PC Says. The program named MENU.BAS is the program-selection menu.

Making Copies of the Diskette

Before using the diskette, it is recommended that you make a copy of it, with the original set aside for safekeeping. The copy can be made either with the DOS commands COPY or DISKCOPY. To create a mirror-image copy of the disk, use the DISKCOPY command as follows. First, format an unused diskette with the DOS command FORMAT. Next, invoke the DISKCOPY procedure as follows:

```
DISKCOPY A: B:
```

Insert the samples diskette in drive A, and your blank disk in B. For single-disk drive users, the system will prompt you when it is time to change diskettes.

To merge the sample programs onto a disk containing some of your own programs, use the COPY command as follows:

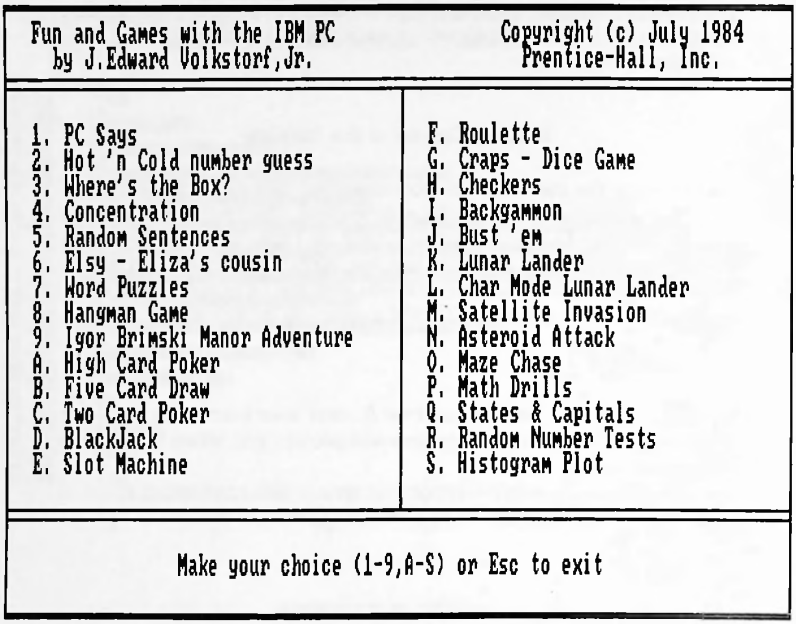
```
COPY A:*.*B:
```

Running the MENU Program

To run a particular sample, you must first be under BASIC. This is done by typing BASICA at the DOS command level prompt A>. Once BASIC is loaded, type RUN "MENU, and a list of the names of all sample programs will be displayed–neatly formatted in two columns on your screen (see Figure B.1). To select one of the programs, simply press the key for the number or letter (1 to 9, and A to S) listed beside the program you want to play. The screen will clear and the program will be loaded from the diskette and started.

Alternatively, the samples can be called directly by typing RUN followed by the program's filename. A quotation mark must precede the name, as in RUN "BACG for the backgammon game. Furthermore, you can also type BASICA MENU to the DOS command A> and start both BASIC and the menu.

FIGURE B.1. Sample of MENU program being used.



Note that the samples diskette menu assumes it has been inserted in disk drive A. Thus when it calls up a game, it always looks for the game program file on drive A. To run the menu in disk drive B requires setting the default disk drive letter to B while in DOS.

Creating an AUTOEXEC.BAT File

A copy of DOS and BASICA can be added to your copy of the samples diskette. This enables creating a disk that starts up automatically with the MENU program when your computer is powered on or rebooted. To do so, first FORMAT an unused diskette with the /S option, as in:

```
FORMAT B: /S
```

which copies the necessary DOS files to the disk after formatting it. Then COPY the system program BASICA.COM (and any other DOS programs you would like, such as DISKCOPY.COM, CHKDSK.COM, and so on). The sample programs can then be copied over, as described earlier, with the COPY command.

Next, the batch file AUTOEXEC.BAT must be created. This need only be a file containing the single record: BASICA MENU. The TIME and DATE commands might also be added, if you desire to have the current time and date on the system. This batch file can be created with the following dialogue on your PC while in DOS:

```
A>COPY CON: AUTOEXEC.BAT
BASICA MENU
^Z
1 file(s) copied
A>
```

where the COPY command is used to copy text entered at the keyboard (or CON: for console) into the file AUTOEXEC.BAT. Press function key 6, and then press Enter for the last record in your file. The system will respond with a^ Z, and conclude the copy procedure. Your batch file is now ready.

Assuming that all the required files are on the disk, when the system is powered on with this diskette, Advanced BASIC will be loaded and the MENU program will automatically start. Make your choice for the game you want to play, and away you go . . .

APPENDIX C: A MENU PROGRAM

The menu program that is provided with the samples diskette simplifies accessing the game programs. The menu lists the descriptive names of each game and allows you to select the game by entering the number or letter listed by the game's descriptive name.

Internal to the menu program, a DATA statement table contains the file names and descriptive names of each game program. It is used to list them on the screen when the menu program is run. After a game is selected, its file name from the table is used in a CHAIN statement to invoke the program. Program C.1 contains the complete listing.

When the menu program begins, it first formats the screen with a number of character mode boxes and displays of various credits

```
0 ' MENU Select game programs
10 SCREEN 0,1:COLOR 7,0:WIDTH 80:KEY OFF:CLS
20 DEFINT A-Z:DIM P$(31),PN$(31)
30 FOR K=1 TO 10:KEY K,"":KEY(K)OFF:NEXT
40 '
50 ' print borders on screen
60 '
70 COLOR 13,0
80 LOCATE 1,2:PRINT CHR$(218);STRING$(74,196);CHR$(191);     ' top
90 FOR R=2 TO 24
100 LOCATE R,2:PRINT CHR$(179);:LOCATE ,77:PRINT CHR$(179); ' sides
110 NEXT
120 LOCATE 25,2:PRINT CHR$(192);STRING$(74,196);CHR$(217);   ' bottom
130 LOCATE 4,2
140 PRINT CHR$(198);STRING$(37,205);CHR$(209);STRING$(36,205);CHR$(181)
150 FOR R=5 TO 20:LOCATE R,40:PRINT CHR$(179);:NEXT ' middle
160 LOCATE 21,2
170 PRINT CHR$(198);STRING$(37,205);CHR$(207);STRING$(36,205);CHR$(181)
180 COLOR 15,12
190 '
200 ' display credits
210 '
220 LOCATE 2,4 : PRINT SPC(73); : LOCATE 3,4 : PRINT SPC(73);
230 LOCATE 2,5 : PRINT"Fun and Games with the IBM PC";
240 LOCATE 2,50: PRINT"Copyright (c) July 1984";
250 LOCATE 3,7 : PRINT"by J.Edward Volkstorf,Jr.";
260 LOCATE 3,52: PRINT"Prentice-Hall, Inc.";
270 '
280 ' list the games
290 '
300 RESTORE 490
310 ML$="123456789ABCDEFGHIJKLMNOPQRS"
320 FOR I=0 TO 27
330 READ P$(I),PN$(I) ' desc. and file names
340 LOCATE (I MOD 14)+6,(I\14)*38+4
350 COLOR 0,11: PRINT " ";MID$(ML$,I+1,1);".";
360 COLOR 3,0 : PRINT USING "\"+SPACE$(27)+"\" ; " "+P$(I)
370 NEXT
380 COLOR 7,0
390 '
400 ' get player's choice
410 '
420 LOCATE 23,19:PRINT"Make your choice (1-9,A-S) or Esc to exit";
430 C=ASC(INPUT$(1)) : IF C=27 THEN CLS:END
440 C$=CHR$(C+32*(C>96)) : PN=INSTR(ML$,C$) : IF PN=0 THEN 430
450 CHAIN PN$(PN-1),20 ' start at line 20 in game program
460 '
470 ' description and filename
480 '
490 DATA PC Says , PCSAYS
500 DATA Hot 'n Cold number guess , HOTNCOLD
510 DATA Where's the Box? , BOXIT
520 DATA Concentration , CONC
530 DATA Random Sentences , RSEN
540 DATA Elsy - Eliza's cousin , ELSY
550 DATA Word Puzzles , WPUZ
560 DATA Hangman Game,HMAN
570 DATA Igor Brimski Manor Adventure , MANOR
```

```
580 DATA High Card Poker , 1CARD
590 DATA Five Card Draw , DRAW
600 DATA Two Card Poker , 2CARD
610 DATA BlackJack , BJACK
620 DATA Slot Machine , SLOTS
630 DATA Roulette , ROUL
640 DATA Craps - Dice Game , CRAPS
650 DATA Checkers , CHECK
660 DATA Backgammon , BACG
670 DATA Bust 'em , BUSTM
680 DATA Lunar Lander ,LEML
690 DATA Char Mode Lunar Lander , CHLEML
700 DATA Satellite Invasion , SATIN
710 DATA Asteroid Attack , ASTROID
720 DATA Maze Chase , MAZE
730 DATA Math Drills , ADRILLS
740 DATA States & Capitals , STATES
750 DATA Random Number Tests , RBAR
760 DATA Histogram Plot , RHISTO
```

Program C.1 MENU - Game selection

and other messages. It then loads from the above-mentioned DATA statement table the game's description into array P$ and the program's name into array PN$.

The two-column display of game names is also done as the arrays are loaded. Next, the program queries the user for a single character response. The character entered by the user is used in an INSTR statement and a string table, ML$, of the single-digit and letter codes that are displayed by each game. The position in this table is an index to the program name array, PN$, which is then used to CHAIN to the selected game. If the position returned by the INSTR statement is zero, it means the user made an invalid selection.

Note that the CHAIN statement in line 450 chains to line 20 of the selected game program. This is to bypass the MENU$ variable flag that tells each game the manner in which it was started. This flag is normally set at line 10 of each program.

APPENDIX D: REFERENCES AND SOURCES FOR MORE INFORMATION

The following list is only a small sampling of the many books available on the subject of computer games, general gaming, probability, statistics, and other related subjects. The books listed below have most influenced this book in one form or another.

BASIC Gaming

The books in this section are generally collections of various BASIC gaming programs, except where noted. The two books by David Ahl are an excellent source for many different games, although all are printer based. Converting many of them to a PC graphics mode should not be too difficult for the diligent reader. It appears that some enterprising companies have already done this, and have marketed the results as original creations.

Ahl, David H., *BASIC Computer Games.* New York: Workman Publishing, 1976.

Ahl, David H., *More BASIC Computer Games.* New York: Workman Publishing, 1979.

Franklin, Howard M. et al, *Golden Delicious Apple Games.* New York: John Wiley and Sons, Inc., 1982.

SYBEX, *Apple PASCAL Games.* Los Angeles: SYBEX Inc., 1982.

On the Subject of Gaming

The following books are useful sources for game ideas, as well as rules and playing strategies for many popular games. Any good-sized bookstore should have many, many more similar titles on its shelves. The Gamblers Book Club, Las Vegas, is another source for innumerable casino and sports gambling books.

Freeman, Jon, *The Playboy Winner's Guide to Board Games.* Chicago: Playboy Press, 1979.

Gibson, Walter B., *Hoyles Modern Encyclopedia of Card Games.* New York: Dolphin Books, Doubleday and Co., 1974.

Gygax, Gary, *Advanced Dungeon & Dragons—Dungeon Master's Guide.* New York: Random House, 1979.

Sackson, Sid, *A Gamut of Games.* New York: Castle Books, 1969.

Advanced and Technical Gaming Books

The books in this category are somewhat technical in nature and generally require anywhere from a minor to extensive mathematical background to understand fully. For the blackjack enthusiast, there are strategies for winning by Edward Thorpe, who first popularized

the technique that was supported by Julian Braun, who now has his own book. For the "prove it to me" individual, there's Peter Griffin's very deep treatise on blackjack. Allan Wilson's book is a good introductory text, and also has an interesting story on finding biased roulette wheels (circa 1955). Richard Epstein reviews all forms of gaming, including gambling, bridge, and chess. The books by Thomas Naylor and Hugh Watson are good medium- to upper-level sources for more information on management games, simulations, and computer models.

Braun, Julian H., *How to Play Winning Blackjack.* Chicago: Data House Publishing Co. Inc., 1980.

Epstein, Richard, *The Theory of Gambling and Statistical Logic.* New York: Academic Press, 1977.

Griffin, Peter, *The Theory of Blackjack.* Las Vegas: Gamblers Book Club Press, 1979.

Naylor, Thomas N. et al, *Computer Simulation Techniques.* New York: John Wiley and Sons, Inc., 1968.

Thorp, Edward O., *Beat the Dealer.* New York: Vintage press, subsidiary of Random House, 1962, 1966.

Watson, Hugh J. *Computer Simulation in Business.* New York: John Wiley and Sons, Inc., 1981.

Wilson, Allan N., *The Casino Gambler's Guide.* New York: Harper & Row, Publisher, Inc., 1965.

INDEX

Abort, key, 2
Action games (*see* Animated action games)
ADRILL S (*see* Arithmetic Drills)
Adventure games, 22, 74–90
  characteristics of, 74–76
  idea of, 74
  (*See also* Igor Brimski Manor Adventure Game)
Animated action games, 20–22, 184–224
  coding images, 184–90
  moving images on the screen, 185–87
  player control of image movement, 188–90
  variations, 223–24
  (*See also* Asteroid Attack; Bust 'em; CHLEMI; LEML; Maze Chase; Satellite invasion)
Arithmetic Drills, 228–34
  complete program listing, 230–33
  example of play, 229
  idea of, 228
  program design, 228–29, 233–34
  variations, 243
ASCII character set, 49, 247
Asteroid Attack, 210–17
  complete program listing, 215–17
  example of, 211
  idea of, 210
  program design, 211–14
  variations, 224
  (*See also* Animated action games)
ASTROID (*see* Asteroid Attack)
AUTOEXEC.BAT file, creation of, 270–71

Baccarat, 129
BAGG (*see* Backgammon)
Backgammon, 168–82
  complete program listing, 175–81
  example of play, 171
  idea of, 168–70
  program design, 170, 172–75
  variations, 181–82
Bargraphs, random number (RBAR), 256–57
BASIC, 246
Bell curve, 260
BJACK (*see* Blackjack)
Blackjack, 92, 102, 122–29
  complete program listing, 125–29
  example of, 123
  idea of, 122
  program design, 122–25
  variations, 129
  (*See also* Card games)
BLOAD statement, 247
Board games, 22, 156–82
  development of, 156–58
  variations, 181–82
  (*See also* Backgammon; Checkers)
Books, 273–75
Boxit, 34–38
  complete program listing, 36–37
  example of play, 35
  idea of, 34
  program design, 34–35, 37–38
  variations, 45–46
BOXIT (*see* Boxit)
Bust 'em, 190–94
  complete program listing, 193–94
  example of play, 191
  idea of, 190
  program design, 190–93
  variations, 223
  (*See also* Animated action games)
BUSTM (*see* Bust 'em)

C, 247
Card games, 21, 22, 92–129
  card image graphics, 95–97
  coding the card deck, 93–94
  computer's play in, 101
  multiplayer, 99–101
  player input and output, 97–99
  purposes of, 92
  shuffling the deck, 94–95
  techniques used in programming, 92–93
  variations, 129
  (*See also* Blackjack; Five-Card Draw; High Card; Two-Card Monte)
Cartridge BASIC, 247

Casino gambling games (*see* Gambling games)
CHAIN statement, 248
Character mode displays, custom design of, 247
Character text, manipulation of (*see* Word games)
CHECK (*see* Checkers)
Checkers, 158–68
  complete program listing, 164–68
  example of play, 159
  idea of, 158
  program design, 158–63
  variations, 181
Chess, 157
CHLEML (Character Mode Lunar Excursion Module Lander), 200–205
  complete program listing, 201–3
  example of play, 201
  idea of, 200
  program design, 200, 204–5
  variations, 223
  (*See also* Animated action games)
CHR$ function, 49, 94
COLOR statement, 184, 185
Computer, role of, 20
CONC (*see* Concentration)
Concatenation, 48
Concentration, 38–45
  complete program listing, 40–42
  example of play, 39
  idea of, 38–39
  program design, 39, 42–45
  variations, 46
Congruential method, 253
Continuous uniform distribution, 258
Conversationalist game (*see* "Eliza-like" program)
COPY command, 269
Craps, 132–33, 146–53
  complete program listing, 148–52
  example of play, 147
  idea of, 146
  program design, 147–48
  variations, 152–53
CRAPS (*see* Craps)
Custom designed programs, 247–48

Delay loops, 189
Discrete uniform distribution, 259
DISKCOPY command, 269
DRAW (*see* Five-Card Draw)
Drill and practice courseware, 226–28
  (*See also* Arithmetic Drills; States and Capitals)

Educational games, 22, 226–43
  types of, 226–28
  (*See also* Arithmetic Drills; States and Capitals)
"Eliza-like" program (ELSY), 52, 55–59
  complete program listing, 57–59
  example of play, 57
  idea of, 55
  program design, 55–56
  variations, 71
ELSY (*see* "Eliza-like" program)
Empirical distributions, 262–63
"Enter," 2
Experimentation, interactive, 227–28

Fact-based programs, 226, 227
  (*See also* States and Capitals)
Five-Card Draw, 102, 107–15
  complete program listing, 110–15
  example of play, 107
  idea of, 107
  program design, 108–10
  variations, 129
  (*See also* Card games)
Flashcard approach, 227
FORTH, 247

Gambling games, 22, 132–53
  (*See also* Blackjack; Craps; Roulette; Slots)
Games, 1
  classification of, 21–22
  creation of, 247–48
  references and sources for, 273–75
  (*See also* Adventure games; Animated action games; Board games; Card games; Educational games; Gambling games; Guessing games; PC Says; Word games)
Gaming programs (*see* Program design; Program listings, complete; Programs)
Graphics mode displays, custom design of, 247, 248
Guessing games, 21, 26–46
  basics of, 26
  variations, 26, 45–46
  (*See also* Boxit; Concentration; Hot 'n' Cold)

Hangman, 67–71
  complete program listing, 69–71
  example of play, 67
  idea of, 67–68
  program design, 68–69
  variations, 72
HCARD (*see* High Card)
High Card, 102–6
  complete program listing, 104–6
  example of play, 103
  idea of, 102
  program design, 102–3
  variations, 129
  (*See also* Card games)
Histograms, random number (RHISTO), 264–68
HMAN (*see* Hangman)
Hot 'n' cold, 27–34
  complete program listing, 31–33
  example of play, 29
  idea of, 27
  program design, 27–31
  variations, 45
HOTNCOLD (*see* Hot 'n' Cold)

Igor Brimski Manor Adventure Game, 76–90
  complete program listing, 84–89
  example of play, 79
  idea of, 76–77
  program design, 77–78, 80–84
  variations, 90
Information references and sources, 273–75
INKEY$ function, 2
Intelligence, artificial, 156–58
Interactive experimentation, 227–28

KEY OFF statement, 189
KEY ON statement, 188–89
KEY STOP statement, 189
Knowledge-based programs, 226, 227
(See also States and Capitals)

Lamba, 260
Learning experience, 20
LEML (Lunar Excursion Module Lander), 195–200
character-mode version (see CHLEML)
complete program listing, 198–200
example of play, 195
idea of, 195
program design, 196–98
variations, 223
(See also Animated action games)
LINE INPUT statement, 2, 98, 99
LOCATE statement, 184, 185
Lunar Excursion Module Lander (see CHLEML; LEML)

MANOR (see Igor Brimski Manor Adventure Game)
MAZE (see Maze Chase)
Maze Chase, 217–23
complete program listing, 220–23
example of play, 219
idea of, 217–18
program design, 218–20
variations, 224
(See also Animated action games)
Menu program, 15, 248–49, 268–73
Multiplayer card games, 99–101
Multiple players, 21

Negative exponential distribution, 260–61, 267
Normal distribution, 266
Null string, 50

ON KEY statement, 188

Pascal, 247
PC Says 6–18
complete program listing, 16–18
example of play, 16
idea of, 6
program design, 6–16
variations, 22–23
PCSAYS (see PC Says)
PEEK statements, 246, 247
Player control, 20–21
Poisson distribution, 262, 268
Poker (see Five-Card Draw; Two-Card Monte)
POKE statements, 246, 247
PRINT statement, 184, 185
Program design:
Arithmetic Drills (ADRILLS), 228–29, 233–34
Asteroid Attack (ASTROID), 211–14
backgammon (BACG), 170, 172–75
Blackjack (BJACK), 122–25
Boxit (BOXIT), 34–35, 37–38
Bust 'em (BUSTM), 190–93
card games in general, 92–93
checkers (CHECK), 158–63
Concentration (CONC), 39, 42–45
craps (CRAPS), 147–48
custom, 247–48
"Eliza-like" program (ELSY), 55–56
Five-Card Draw (DRAW), 108–10
Hangman (HMAN), 68–69
High Card (HCARD), 102–3
Hot 'n' Cold (HOTNCOLD), 27–31
Igor Brimski Manor Adventure Game (MANOR), 77–78, 80–84
Lunar Excursion Module Lander (LEML), 196–98
Lunar Excursion Module Lander, charactermode version (CHLEML), 200, 204–5
Maze Chase (MAZE), 218–20
programming languages, 20, 246–47
random sentence generator (RSEN), 51–52
roulette (ROUL), 140–43
Satellite Invasion (SATIN), 205–7, 210
slots (SLOTS), 133–36
States and Capitals (STATES), 235, 239–43
Two-Card Monte (2CARD), 116–17
word-find puzzle (WPUZ), 60–63, 65–67
Program listings, complete:
Arithmetic Drills (ADRILLS), 230–33
Asteroid Attack (ASTROID), 215–17
backgammon (BACG), 175–81
Blackjack (BJACK), 125–29
Boxit (BOXIT), 36–37
Bust 'em (BUSTM), 190–94
checkers (CHECK), 164–68
Concentration (CONC), 40–42
craps (CRAPS), 148–52
"Eliza-like" program (ELSY), 57–59
Five-Card Draw (DRAW), 110–15
Hangman (HMAN), 69–71
High Card (HCARD), 104–6
Hot 'n' Cold (HOTNCOLD), 31–33
Igor Brimski Manor Adventure Game (MANOR), 84–89
Lunar Excursion Module Lander (LEML), 198–200
Lunar Excursion Module Lander, charactermode version (CHLEML), 201–3
Maze Chase (MAZE), 220–23
PC Says (PCSAYS), 16–18
random number bargraphs (RBAR), 256–57
random number histograms(RHISTO), 264–65
random sentence generator (RSEN), 53–54
roulette (ROUL), 143–46
Satellite Invasion (SATIN), 207–9
select game programs (MENU), 272–73
slots (SLOTS), 136–39
States and Capitals (STATES), 236–39
Two-Card Monte (2CARD), 117–21
word-find puzzle (WPUZ), 63–65
Programming languages, 20, 246–47
Programs:
basic information on, 2–3
common attributes of, 19–21
menu, 15, 248–49, 268–73
(See also Program design: Program listings, complete)
Progressive levels of play, 19–20
Prompt processor, 243
PUT statement, 185–87

Queuing theory, 261

RANDOMIZE statement, 253–55
Random numbers:
  continuous uniform distribution, 258
  discrete uniform distribution, 259
  empirical distribution, 262–63
  generation of, 251–57
  miscellaneous distributions, 263
  negative exponential distribution, 260–61, 267
  normal distribution, 260, 266
  plotting, 264–68
  Poisson distribution, 262, 268
  random number bargraphs (RBAR), 256–57
  random number histograms (RHISTO), 264–68
Random sentence generator (RSEN), 50–54
  complete program listing, 53–54
  example of play, 53
  idea of, 50–51
  program design, 51–52
  variations, 71
RBAR (*see* Bargraphs, random number)
References, 273–75
Repetitive skill tasks, 226
  (*See also* Arithmetic Drills)
RHISTO (*see* Histograms, random number)
RND function, 95, 101, 251–55
Role-playing games (*see* Adventure games)
ROUL (*see* Roulette)
Roulette, 132, 139–46
  complete program listing, 143–46
  example of play, 140
  idea of, 139–40
  program design, 140–43
  variations, 152–53
RSEN (*see* Random sentence generator)
RUN command, 2

Sample-programs diskette, use of, 268–71
Satellite Invasion, 205–9
  complete program listing, 207–9
  example of play, 206
  idea of, 205
  program design, 205–7, 210
  variations, 224
  (*See also* Animated action games)
SATIN (*see* Satellite Invasion)
Scoring, 20–21
Screen-generator software, 247
Screen layout, 7
Select game programs (MENU), 272–73
Selective pruning, 158
Simon Says (*see* PC Says)
Skill, 19
Slots, 132–39
  complete program listing, 136–39
  example of play, 134
  idea of, 133
  program design, 133–36
  variations, 152, 153
SLOTS (*see* Slots)
Sound effects, 19
Speed of game, 20
STATES (*see* States and Capitals)
States and Capitals, 234–43
  complete program listing, 236–39
  example of play, 235
  idea of, 234–35
  program design, 235, 239–43
  variations, 243
Strategy, 19
Strings, manipulation of, 49–50

Timing of games, 20
Tutorial courseware, 226–28
21 (*see* Blackjack)
Two-Card Monte, 102, 115–21
  complete program listing, 117–21
  example of play, 116
  idea of, 115
  program design, 116–17
  variations, 129
  (*See also* Card games)
2CARD (*see* Two-Card Monte)

Uniform distributions, 257–58

VAL function, 2, 254

War, 129
Word-find puzzles, 60–67
  complete program listing, 63–65
  examples of play, 60–62
  idea of, 60
  program design, 60–63, 65–67
  variations, 71–72
Word games, 21–22, 48–72
  processing character data, 48–50
  variations, 71–72
  (*See also* "Eliza-like" programs; Hangman; Random sentence generator; Word-find puzzles)
WPUZ (*see* Word-find puzzles)

Announcing!

Save many hours of tedious typing by buying your copy of the Fun & Games Programs on diskette!

Includes all 28 programs plus menu—all for only $19.95

ORDER TODAY!

Use This Order Form

Please send me the following:

Title	Price	Quantity	Total
Fun and Games BASIC programs disk Single Sided/Double Density, 5 inch, IBM PC format	$19.95		
Graphics Programming on the IBM Personal Computer, book	$16.95		
book/disk	$34.95		
SUBTOTAL			
IF C.O.D. ADD HANDLING $1.50			
VIRGINIA RESIDENTS ADD 4% SALES TAX			
GRAND TOTAL			

METHOD OF PAYMENT:

☐ Check ☐ C.O.D. Charge My: ☐ VISA ☐ MasterCard

Cardholder's Name _____

Card Number _____

Exp. Date _____

SHIP TO _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

MAIL TO:

J. Edward Volkstorf, Jr.
Fun & Games Programs
P.O. Box 7882
Charlottesville, VA 22906

WHO NEEDS ARCADES

. . . when you have a software sampler of 28 games right here? Easy to key in but challenging to play, the games programs in this book are *more* than mere entertainment. Fun, yes—but they teach you game design and BASIC programming, too! (Not to mention basic but crucial programming concepts in graphics, animation, random numbers, and statistics.)

Here are all your favorites—like Backgammon and Asteroids—plus many new ones—all brought from the table to the screen:

- board games
- mazes
- classic card games
- adventure games
- space games
- educational games
- and more.

These pastimes incorporate all the *professional* features you've come to expect—such as progressive levels of difficulty for differing abilities. Plus, you can play by yourself or pit your skills against your friends', because the collection features both one- and multiple-player games.

J. Edward Volkstorf, Jr., is a freelance consultant specializing in microcomputer-based educational applications. Previously he spent ten years in the field of hospital financial and clinical data processing as manager of programming for a computer service bureau.

P

ISBN 0-13-332487-7

0 21898 33248 6

Cover design by Hal Siegel
Cover photo by Michel Tcherevkoff
PRENTICE-HALL, Inc.
Englewood Cliffs, New Jersey 07632

Bookware® is a registered trademark of Prentice-Hall, Inc.